USENIX

SAN DIEGO CONFERENCE PROCEEDINGS

WINTER

1989

USENIX

CONFERENCE PROCEEDINGS

San Diego, California
January 30 - February 3, 1989

USENIX Association

Proceedings of the Winter 1989 USENIX Conference

January 30 — February 3, 1989
San Diego, California USA

For additional copies of these proceedings contact
USENIX Association
P.O. Box 2299
Berkeley, CA 94710 USA

The price is $30.
Outside the U.S. and Canada please add
$25 per copy for postage (via air printed matter)

TABLE OF CONTENTS

PLENARY SESSION

Wednesday (9:00-10:30) **Chair: Peter Salus**

RPC AND DISTRIBUTED SYSTEMS

Wednesday (11:00-12:30) **Chair: Lori Grob**

FILE SYSTEMS

Wednesday (2:00-3:30) **Chair: Melinda Shore**

OPERATING SYSTEMS I

Wednesday (4:00-5:30) **Chair: Avadis Tevanian**

WINDOW SYSTEMS

Thursday (9:00-10:30) **Chair: Jeff Schwab**

SPECIAL INTEREST I

Thursday (11:00-12:30) **Chair: Andrew Hume**

INTERNETWORKING

Thursday (2:00-3:30) **Chair: Thomas Narten**

OBJECTS & MEMORY

Thursday (4:00-5:30) **Chair: Jim Lipkis**

WORK IN PROGRESS

Thursday (4:00-5:30) **Chair: Keith Bostic**

VIRUS, WORM AND OTHER UNIX PESTS

Thursday (8:00-10:00) **Chair: Keith Bostic**

SPECIAL INTEREST II

Friday (9:00-10:30) **Chair: Donn Seeley**

OPERATING SYSTEMS II

Friday (11:00-12:30) **Chair: Jon Kepecs**

PROCESSES

Friday (2:00-3:30) **Chair: Jim McGinness**

SECURITY

Friday (4:00-5:30) **Chair: Rick Lindsley**

PREFACE

Welcome to the Winter 1989 USENIX Conference in San Diego! We have selected 36 papers in 11 sessions from a combination of submitted abstracts and full papers. While there is no specific conference theme this time, we do expect a lot of formal and informal discussion of computer virus and worm effects on the Unix environment. Due to the tremendous amount of publicity and controversy raised by the November worm software which spread throughout the Internet, we decided at the last minute to add the special session "Virus, Worms, and Other Unix Pests" on Thursday evening. We suspect that the combination of Tom Duff's engaging paper and this session of professional exterminators will lead to many lively engagements. We have received several very good "worm tour" papers and have choosen one for the proceedings. Thanks to all of you who have helped to make this last minute session possible!

The winners of the student paper award are Charles Bergan and Bruce Martin, coauthors of an excellent paper on Remote Procedure Calls entitled "Experiences with PARPC". The work was done under the sponsorship of the Gemini Project at UCSD, a program under the direction of Professors Walter Burkhard and Jehan-Francois Paris and funded by the University of California Micro Program and the NCR Corporation. The paper will be presented at 11:00 on Wednesday, February 1st. Congratulations Charles and Bruce!

We would like to take this opportunity to thank all of those who helped put this conference together. At the top of the list belong the Program Committee members who were forced (with reluctance) to deal with a very short paper evaluation turn-around period before coming to San Diego for the Program Committee meeting. Judy DesHarnais and Peter Salus also deserve recognition for always being there when they were needed and consenting to answer even the most bizarre questions we could think up! We would also like to thank those who provided their support and expertise as advisors, critics, and reviewers, including past Program Chairs, the paper reviewers, and the many folks behind the scenes who took the time to express their views. Finally our thanks go to Dean M. Lea Rudee, Division of Engineering, and Professor Don Anderson, Director, Office of Academic Computing at UCSD for allowing us the time and resources for our participation in this conference.

We hope you enjoy the conference!

Greg Hidley & Keith Muller
Technical Program Chairs

CONFERENCE COMMITTEE

CONFERENCE ORGANIZERS

Greg Hidley, *Technical Program Chair*
(Division of Engineering, UC San Diego)
Keith Muller, *Technical Program Chair*
(Office of Academic Computing, UC San Diego)
Judith F. Desharnais, *Meeting Planner*,
(USENIX Association)
John Donnelly, *Tutorial Coordinator*
(USENIX Association)
Evi Nemeth, *Proceedings Production*
(University of Colorado, Boulder)

TECHNICAL PROGRAM COMMITTEE

Rick Adams (Center for Seismic Studies)
Keith Bostic (UC Berkeley)
Todd Brunhoff (Tektronix)
John Chambers (University of Texas, Austin)
Lori Grob (NYU)
Greg Hidley, *Chair* (UC San Diego)
Andrew Hume (ATT Bell Labs)
Keith Muller, *Chair* (UC San Diego)
Thomas Narten (Purdue University)
Don Seeley (University of Utah)
Melinda Shore (Frederick Cancer Research Facility)
Eugene Spafford (Purdue University)
Henry Spencer (University of Toronto)
Avadis Tevanian (NeXT)
Karen White (Pyramid)

PROCEEDINGS PRODUCTION

Evi Nemeth, Dotty Foerst, Trent Hein,
Paul Kooros, Laszlo Nemeth, and Tyler
Stevens (University of Colorado)

TECHNICAL PROGRAM REVIEWERS

Hiralal Agrawal
Eric Allman
Jim Blondeau
Mark Bradakis
Shirley V. Browne
Gary Buck
Steve Chapin
Guy Cherry
Geoff Collyer
Donna Converse
Eric C. Cooper
Mike Cripps
Prasun Dewan
H. E. Dunsmore
Mike Edelman
Jan Edler
Bob Eifrig
James Ellis
Beverly Erlebacher
Jeff Forys
Jim Frew
Don Fussell
David Garlan
Paul Gilliam
Jeff Glover
Russ Gorby
Allan Gottlieb
Rich Hammons
Scott Hennes
Guy Harris
Mike Hibler
David Hitz
Clyde Hoover
William Hsu
Janaka Jayawardena
Richard Kenner
Jon Kepecs
Steven Kleiman
Rick Krull
Tracy LaQuey
Terry Laskodi
Jay Lepreau
Guoben Li
Rich Lindsley
James S. Lipkis
Cynthia Livingston
Robert B. Lyon
Mark Mason
Kirk McKusick
Shawn D. Ostermann
Thomas Palmer
Clyde Poole
Lauri Rathmann
Dan Reading
John T. Riedl
Marshall T. Rose
Russel Sandberg
Christine Scherbert
Dave Schiferel
Edith Schonberg
Jeffrey R. Schwab
Spencer Shepler
Preeti Shrikhande
Susanne Smith
Rik Smoody
Leigh Stoller
Carl Sutton
Bob Toole
Daniel Trinkle
Eric Vasilik
Kathy Vinyard
Craig E. Wills
Rajendra Yavatkar

AUTHOR INDEX

Experience with PARPC †

Bruce Martin ††
Charles Bergan
Walter Burkhard
Jehan-François Pâris †††

Computer Systems Research Group
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, California 92093

ABSTRACT

PARPC provides an interprocess communication mechanism based on the semantics of a procedure call. PARPC programs always execute a single logical thread of control but may execute multiple physical threads of control. PARPC provides users with a well defined, high level network process model of execution and a familiar program development model supporting heterogeneous, non-uniform environments. The administrative overhead of PARPC is minimal because users administer their own distributed programs and existing UNIX mechanisms for access control and resource accounting are utilized. Our experiences indicate that PARPC has been an effective system for the development and administration of distributed programs.

1. Introduction

Many distributed algorithms require the capability of sending a message to a set of destinations and getting answers back from some or all of them. Examples of theses algorithms are replicated data consistency protocols, such as majority consensus voting [Giff79] and available copies protocol [Bern83], load balancing algorithms, commit and locking protocols, parallel searches through multiple databases and many distributed software maintenance tasks. The remote procedure call [BiNe84] does not not lend itself well to express these semantics as it can only model interactions between a single client and a single server [TaRe85].

The *parallel procedure call* was developed to overcome this limitation [MaBeRu87, Saty86]. A parallel procedure call allows a client process to request the parallel execution of the same procedure in *n* different address spaces in parallel.

We present in this paper our experience in designing and using the PARPC system [MaBeRu87], a parallel remote procedure call system developed at the University of California, San Diego. The PARPC system came about as a result of the development of the Gemini file system testbed [BuMaPa87]. Gemini was built for experimenting with protocols maintaining the consistency of replicated files. While writing the first version of Gemini, we found that producing code required for communication between machines dominated our development time. The implementation of remote connections, authentication, remote processes initiation and

† This work was sponsored in part by grants from the U.C. MICRO Program and NCR Corporation.

†† Author's current address: Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, California 94304.

††† Author's current address: Department of Computer Science, University of Houston, 4800 Calhoun Road, Houston, TX 77204-3475.

parameter transmission constituted the bulk of the programming effort and hindered us in our efforts to experiment with distributed algorithms.

We decided to develop the PARPC system to directly support parallel procedure calls and a high level model of distributed computation. Furthermore, we decided the system should support program development, execution and administration in the complex distributed computing environment we were experiencing in the Department of Computer Science and Engineering at UCSD.

Our distributed computing environment is complex because it consists of heterogeneous architectures, operating systems, communication protocols and non-uniform file systems. The environment includes a VAX 11/780, a Pyramid 90X, dozens of ATT 3b2 systems, an ATT 3b15, a Celerity 1260D, a CCI 6/32, several subnets of SUN workstations and an NCR Tower. While all of these machines are physically connected through several ethernets, developing and maintaining a distributed program in such a complex environment is tedious at best. A construct like the V kernel process group, which supports multicasting in an homogeneous environment [ChZw85], would not apply.

Throughout this paper we discuss how PARPC bridges complex distributed computing environments. We will refer to a network as *homogeneous* if it contains a single machine architecture *and* a single operating system. Otherwise, it is *heterogeneous*. We classify a file name space as *uniform* if files are globally available through file system calls *and* named the same from all sites in the network.

Our overall distributed computing environment is definitely heterogeneous and non-uniform. However, within this environment, we have homogeneous, uniform subnets (e.g. SUN 3 workstations with uniformly mounted NFS), heterogeneous and uniform subnets (e.g. VAX and SUN systems running NFS) and homogeneous, non-uniform subnets (e.g. ATT 3b2s with local file systems). Thus, we wanted PARPC to gracefully exist in these subenvironments as well. Since the heterogeneous, non-uniform case is the most general environment, the mechanisms that PARPC provides to support it automatically accommodate the other environments.

PARPC supports heterogeneous, non-uniform environments by extending the simple, well understood UNIX model for local program development and execution and system administration. Because the system uses existing UNIX mechanisms to extend the model, our experience has been that distributed program development and system administration is easy.

1.1. Status of the PARPC System

PARPC has been ported and PARPC programs execute on most of the systems in our complex environment. We have ported PARPC to the VAX 11/780 running 4.3 BSD, the Pyramid running OSx, the Sun 3 series running SunOS, the Celerity 1260D running 4.3 BSD, AT&T 3b2 systems running System V and HP Series 300 systems running HP-UX. In addition, PARPC has been distributed to approximately fifteen research and educational institutions.

The implementation of a wide range of distributed programs using PARPC demonstrates the ease of program development and the applicability of the parallel remote procedure call abstraction. We have used PARPC to implement the nested object scheduler described in [Martin87] and [Martin88]. PARPC was used to implement the replicated block server described in [CaLoPa87]. Moreover, the programming and administration tools provided with the system are themselves PARPC programs. These tools are described below.

We now describe our experiences with the PARPC system. In section two we review related work. Section three describes the PARPC programmer model and shows how a typical program would be built using PARPC. Section four describes the administrative requirements of the PARPC system, and section five discusses performance of the PARPC system. More technical information regarding PARPC can be found in [Martin86] and [MaBeRu87].

2. Related Work

Parallel procedure call is similar to replicated procedure call. [Coop85] Since parallel procedure call provides the calling code with control over processing the results, it is a more general construct for expressing distributed algorithms. Traditional remote procedure call was implemented in the Xerox Cedar environment by Birrell and Nelson. [BiNe84] Since then, several remote procedure call systems have been implemented for UNIX including Courier [Coop83] and Sun RPC. [Sun 84b] Remote procedure call provides a model of computation that is limited to both a single *logical* and *physical* thread of control. It is an inadequate model for expressing many distributed algorithms, such as those previously mentioned.

As a remote procedure call system, PARPC is unique in three ways. First, PARPC programs execute a single logical thread of control but may execute multiple physical threads of control. PARPC programs *appear* to be sequential. Secondly, PARPC provides users with a well-defined, high level network process model of execution and a familiar program development model. Finally, the system supports program development in heterogeneous, non-uniform environments.

3. A Programmer's View of PARPC

We have found that fairly unsophisticated C programmers have been able to quickly develop distributed programs using PARPC. We believe this is due to the simplicity of the parallel procedure call construct, the high level model of computation presented to the programmer and the set of UNIX-like development tools provided with PARPC. We discuss each in turn.

3.1. A Parallel Procedure Call

A parallel procedure call is a structured programming construct that is easy for a programmer to understand. Although there may be several *physical* threads of control in a program making parallel procedure calls, there is always a single *logical* thread of control.

A parallel procedure call executes a procedure in *n* different address spaces in parallel. The calling code remains blocked while the *n* procedures execute. The order in which the *n* procedures complete is nondeterministic. As each procedure result becomes available, the caller is unblocked to execute a statement to process the result of the returned call. After executing the statement, the caller reblocks to wait for the next result. This continues until the caller breaks out of the parallel procedure call or until no further results are available.

A parallel procedure, *parproc()*, is invoked from a C program as follows:

`parproc` (*hl, parameters...*) *result statement;*

The first parameter, *hl*, identifies the set of hosts where the procedure is to be executed. PARPC programs manipulate host names as human readable strings.[1] The *result statement* may be a compound statement and may contain `continue` or `break` statements. However, if a `break` is executed, all unprocessed results from the parallel procedures become unavailable.

All data are passed and returned via the parameters. Parameters are designated as either in or out parameters and have copy in and copy out semantics. Since a procedure with no out parameters cannot return data, the calling code cannot depend on results of the remote procedure. In such a case, the calling code does not block and *result statement* is never executed; the parallel remote procedure call is effectively a multicast. [MeBo76]

PARPC programs may have multiple physical threads of control only if there are no logical dependencies between the threads. In particular, procedures executing in parallel cannot

[1] The system provides a library of operations to manipulate lists of hosts. PARPC provides no mechanism for transparently deciding which hosts to use. Such mechanisms could easily be added on top of PARPC. The example in figure 4 assumes the hosts come from the user.

communicate with each other. Furthermore, the calling code can proceed in parallel with the parallel procedures only if it does not depend on any of their results. These semantics make PARPC programs appear sequential, allowing parallel execution in a very controlled fashion.

The parallel procedure call construct is the only mechanism in which a programmer must consider the distributed nature of the program. PARPC programmers are never required to be concerned with connection and authentication protocols, scheduling, data conversion and remote process initiation. These tasks are appropriately handled and hidden by system software. PARPC accomplishes this by providing programmers with a uniform, high-level computation model called a *network process*.

3.2. The Network Process Model of Computation

A network process is a distributed tree of local processes (Figure 1). The root, called the *client*, initiates the computation. Internal nodes, called *servers*, execute procedures and return results. Each node, being a local process, has state. However, local state may only be communicated between nodes via parallel procedure calls. All communication is done between levels; servers cannot communicate among themselves. However, servers may themselves be clients of other servers.

Network processes retain local process semantics. UNIX process operations, such as *fork(), execve()* and *exit()* are analogously defined for the network process. Network process integrity is assured by the system. As with local processes, access to resources is controlled for network processes.

If a machine fails in the network process, the subtree rooted at the failed machine terminates. The code making the parallel procedure call can detect the failure in the result statement but need not be concerned with orphaned servers. Similarly, if the client fails, servers terminate automatically.

PARPC supports two server models: *user* servers and *resident* servers. User servers are simpler to use but do not provide the flexibility of the resident server. The user server model allows ordinary, unprivileged users to develop and execute network processes. User server code is automatically initiated by the PARPC system. Network process semantics and integrity are ensured even though the user has no special privileges. The resident server model, on the other hand, allows privileged resident applications, such as file servers, to be easily constructed. Server nodes of resident applications are not automatically initiated by the PARPC system and are given control over the authentication process. The type of server used is a link-time decision and has no impact on the application code. The PARPC server used to invoke users servers is itself implemented as a resident server.

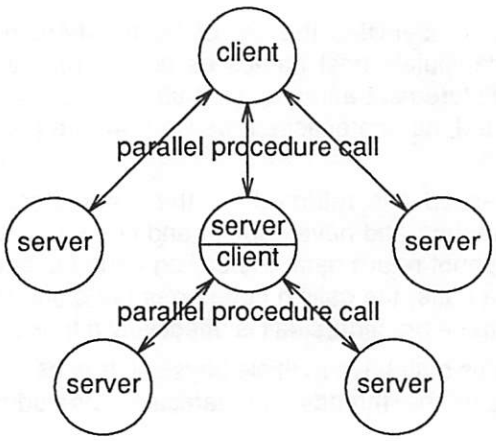


Figure 1: Example Network Process

3.3. PARPC Program Development

PARPC program development easily integrates into the UNIX environment. Building a PARPC program consists of the familiar and simple steps of building a UNIX program. PARPC programs can be built incrementally. Like UNIX programs, PARPC programs are bound together statically. Global consistency is achieved using existing UNIX tools, such as *make* [Feld79] and *lint* [John79]. New tools provided with the system have familiar UNIX-like interfaces.

Many remote procedure call mechanisms, such as Sun RPC [Sun84a] and the Unix implementation of Courier [Coop83], utilize a distinct language to represent externalized data. This forces a program to manipulate both data representations and a programmer to specify a program in more than one language. PARPC programmers specify the interface between clients and servers using a subset of C or C++[KeRi78][Stro86]. The interface is given in a header file as a set of type declarations, procedure declarations and procedure argument specifications. Since the syntax of the interface is designed to be compatible with C and C++ syntax, the same specification file can be used by the compiler or lint to check types of parameter lists of procedure calls and definitions.

Header files containing data type and procedure declarations are processed by a program called *cstub*. *Cstub* produces object code (called *stubs)* that converts data between local and network representations and calls routines in the PARPC run time libraries. Stubs are linked with both the application code and the run time libraries to produce the desired PARPC program.

Cstub statically binds client code to server code by storing a network wide unique identifier, called the *cookie*, in both the client and server stubs. To ensure consistency of client and server code, the cookie is dynamically verified when a node of the network process is created.

The C source files produced by *cstub* are are automatically compiled to produce the following three files:

`client.o`, an object module containing the client stubs,

`server.o`, an object module containing the server stubs,

`defs.h`, a header file containing redefinitions of the parallel procedure calls.

PARPC programs are created in three simple steps. *Cstub* transforms interface specifications into client and server stubs. Application code is compiled. Finally, each node of the PARPC program is created by linking application code, the stubs and the run time libraries together.

For environments where all hosts share a uniform file name space and where each host has the same machine architecture and operating system, *cstub* and the C compiler are sufficient for building PARPC programs. All source code resides in a common location and there is just one copy of all executable programs. In heterogeneous, non-uniform environments, the executable code must reside at each machine. *Cstub* and the C compiler must be executed on each machine in the system. To facilitate this, PARPC provides the replicated compilers, *rcstub* and *rcc*. These tools are themselves implemented as PARPC programs and execute as network processes. Through the use of these tools, source code can be stored in a single location. These tools provide a clean and easy-to-use environment for PARPC program development and maintenance. Figure 2 shows the development steps for four possible distributed environments.

3.4. An Example

We demonstrate the simplicity of the tools with a non-trivial example. We demonstrate the construction of a distributed program to balance the load of a set of printers. The client portion of the program first queries a set of hosts for the status of their respective printers. The client sends the data to the least busiest printer.

	Uniform File Name Space	Non-uniform File Name Space

Homogeneous Architecture and OS	`cstub intf.h`
	`cc client`
	`cc server`
	`mkserver server`
		`rcstub <machines> intf.h`
		`rcc <machines> client`
		`rcc <machines> server`

Heterogeneous Architecture or OS	`rcstub <machines> intf.h`
	`rcc <machines> client`
	`rcc <machines> server`
		`rcstub <machines> intf.h`
		`rcc <machines> client`
		`rcc <machines> server`

Figure 2: PARPC Development Support.

Figure 3 gives `printservops.h`, the interface between the client and the printer servers.

```
typedef struct {
    char *data;
    int data_length;
} string;

remote getprinterload (/* hostlist, out int *load */);

remote printfile (/* hostlist, in string filename, out int *error */);
```

Figure 3: `printservops.h`

This file is input to *cstub*. The interface language is a subset of C that is extended with the keywords `remote`, `in` and `out`. Cstub generates client and server stubs for the procedures prefixed by remote. When the C compiler parses the header file, `remote`, `in` and `out` are hidden by the C preprocessor. Other than the `in` and `out` keywords, parameters can be specified using either C++ or ANSI C syntax. Thus, compilers that support type checking of parameters ensure consistency between clients and servers. However, in order to support older C compilers that do not allow parameter specifications, the interface language allows parameters to be specified between C comment symbols `/*` and `*/`.

Figure 4 gives `print.c`, the client algorithm. After building a list of hosts from the command line, the parallel procedure, `getprinterload()`, is executed to find a lightly used printer. As each procedure result becomes available, the statement following the call to `getprinterload()` is evaluated using the value of `load` set by the remote procedure. If no failures occurred and an unused printer was found, the `addfrom` operation sets `printhl` to reference the unused printer. The client then breaks out of the parallel procedure call. The PARPC system discards all further responses about other printers. Finally, if a printer was indeed found, the file is printed by executing `printfile()` on the host referenced by `printhl`. Such a parallel procedure call made to a single host is a traditional remote procedure call.

Figure 5 gives `printserv.c`, the server algorithm.

We now demonstrate how to construct this distributed program in a homogeneous, uniform environment. An example of this environment is a network of architecturally compatible SUN workstations with uniformly mounted NFS. These steps could easily be integrated into a *make* file.

First the client and server stubs, `client.o` and `server.o`, are created as follows:

```
cstub printservops.h
```

Cstub automatically places a network wide unique tag, called the *cookie*, in both `client.o` and `server.o` to ensure consistency between clients and servers. Now the application files are compiled.

```
cc -c print.c printserv.c
```

```
#include <par_rpc.h>
#include <defs.h>
#include printservops.h

main(argc,argv)
{
    hostlist queryhl, printhl;
    int load, error, minload = MAXINT;
    /* Parse arguments and build host list for querying printer loads */
        :
        :
    /* Parallel procedure call to get printer load from each
       printer given in queryhl.  The expression evaluated for
       each response finds the least loaded printer and sets
       printhl to reference that printer.
    */
    getprinterload(queryhl,&load)
        if (!host_error(queryhl)) { /* if no system error */
            if (load == 0) {  /* found an unused printer, use it. */
                clearhl(printhl);
                addfrom(printhl,queryhl);
                break;
            }
            if (load < minload) {  /* found a less used printer, consider it */
                clearhl(printhl);
                addfrom(printhl,queryhl);
                minload=load;
            }
        }

    if (emptyhl(printhl)) {
        printf("No printers available\n");
        exit(1);
    }

    /* Remote procedure call to send file name to printer
       referenced by printhl. Host_error() indicates whether a system
       failure occurred;  error returns an application error code.
    */
    printfile(printhl,argv[1],&error)
        if (host_error(printhl) || error!=0)
            printf("Error %d printing %s",error,argv[1]);
}
```

Figure 4: `print.c`

Next the client executable program is created. The `-l` option tells the loader to use the PARPC client runtime library.

```
cc -o print print.o client.o -lclient
```

Finally, the server executable program is created. The `-l` option tells the loader to use the PARPC user server runtime library.

```
cc -o printserv printserv.o server.o -lserver
mkserver printserv
```

Files containing executable server nodes are automatically invoked by the parent PARPC server. The `mkserver` program lets the system know about the new server by creating links to the server file. The link is named by the directory where the parent PARPC server executes and the unique cookie. Users own and control access to server files using standard UNIX file access modes. Since client nodes identify server nodes indirectly via the cookie, users may rename files containing server nodes without problems.

```
#include <par_rpc.h>
#include printservops.h

remote getprinterload (qhl, load)
    hostlist qhl;
    int *load;
{
    /* return current load for local printer */
}

remote printfile (phl, filename, error)
    string filename;
    int *error;
{
    /* print file named by filename on local printer */
    /* set error if trouble occurred printing the file */
}
```

Figure 5: `printserv.c`

Next we demonstrate how to construct this same distributed program in a heterogeneous, non-uniform environment. We will construct the program for a Pyramid system, called *beowulf*, for an ATT 3B2 system, called *ishtar* and for a subnetwork of SUN workstations. The SUN workstations form a homogeneous and uniform subnet; therefore, we need only construct it using a single machine, called *napoli*. The PARPC program will be available from all of the SUN workstations in the subnet.

First the client and server stubs are created around the network as follows:

```
rcstub beowulf:/usr/print/src ishtar:/local/src napoli: printservops.h
```

Rcstub invokes *cstub* on hosts beowulf, ishtar and napoli. *Cstub* creates `client.o` and `server.o` in directory `/usr/print/src` on beowulf, in `/local/src` on ishtar and in the current directory on napoli. Each host is sent a copy of `printservops.h`, as well as any non-system *include* files. To accommodate differences in heterogeneous environments, host-specific *cstub* options may be specified on the command line.

Rcstub automatically generates a unique network-wide cookie and sends it as a argument to each remote invocation of *cstub*. The cookie globally defines the interface around the network.

Next the application code is compiled around the network as follows:

```
rcc -c beowulf:/usr/print/src ishtar:/local/src napoli: print.c printserv.c
```

Rcc invokes the C compiler around the network. As with *rcstub*, *rcc* sends all necessary source files to each host where the compilation is to take place. Arguments to *rcc* are a list of hosts, host-specific *cc* options and default *cc* options that are applied to all hosts. *Rcc* may be used to build non-distributed UNIX programs as well.

With the appropriate options, *rcc* may be used to invoke the linker around the network to produce executable PARPC programs. Thus, the client and server executable files are created around the network as follows:

```
rcc -o print beowulf:/usr/print/src ishtar:/local/src napoli: print.o client.o -lclient
rcc -o printserv beowulf:/usr/print/src ishtar:/local/src napoli: printserv.o server.o -lserver
```

To inform the PARPC system about the existence of a new server, `rcc` automatically invokes the `mkserver` program around the network.

4. An Administrator's View of PARPC

A remote procedure call package affects two areas of system administration: security and system resources. It must ensure that a remote user cannot gain access to any resources to which he would normally be denied, and that any resources used by a remote process are charged to him. UNIX provides simple and elegant mechanisms for both access control and accounting in a local environment. Our goal was to extend the use of these mechanisms in a heterogeneous, non-uniform environment. We designed a simple solution. PARPC uses a translation table which ensures all processes created by a remote user are run as a local equivalent. This allows the standard UNIX accounting and access control facilities to be used. By supporting the use of standard UNIX facilities on PARPC programs, the additional administrative overhead of PARPC is minimal.

4.1. Access control

Access control mechanisms restrict the access capabilities of remote processes. PARPC extends the UNIX access control mechanisms by propagating user and group identifiers across hosts. This poses a problem in a heterogeneous network environment since user identifiers are not necessarily consistent across machines. PARPC provides a translation mechanism which, given a host and a user identifier, returns the local representative of the user. These equivalencies are given in an installation-supplied network equivalency file which lists equivalent users on different hosts. For example, on our local computer network, this equivalency file contains the line:

424@napoli 424@roma 424@thor 417@ishtar 419@beowulf

A network process whose client was created by a user whose (effective) uid is 417 on ishtar is given (effective) uid 424 on napoli. PARPC also supports the idea of a *network group*. A network group identifier is network-wide unique integer representing a group existing on all machines. PARPC allows the administrator to specify a global constant called the *minimum group id*. Any groups to which the calling procedure belongs which are above this constant are propagated to the remote process. This allows for the creation of network wide groups.

This translation scheme requires the secure transmission of the caller's credentials. We have implemented this secure transmission through the use of a program called *start*. *Start* is responsible for ascertaining the user id, effective user id, group list (BSD) and effective group id (System V) of the caller and transmitting them to the server. Start itself is a privileged UNIX program and transmits this information to the host via a root only socket. By using a root only socket, we ensure that no user process can transmit a false identity to the remote host. The start program tells the remote process which port the caller will be using. This enables the remote process to verify that no user can surreptitiously break into a communication and gain the access permissions of the caller.

Since the UNIX access control mechanism is utilized, clients or servers may be setuid programs. This allows PARPC programs to be executed by users for which no local equivalent exists on remote hosts.

4.2. Server Files

To initiate a remote service, the client must somehow identify the server code he wishes to use. Storing absolute path names in client programs is inappropriate as file path names can change. Furthermore, in a heterogeneous or non-uniform environment, absolute path names for servers must differ. Therefore, indirect server referencing is a requirement of remote procedure call systems.

The PARPC system has two conflicting goals with regards to the placement of server files: indirect server referencing and user control. One way to implement indirect server referencing is to force the servers to be placed in a specific directory on each machine. Such an approach is used by Courier.[Coop83] However, a common shared directory must be maintained by a system administrator. Furthermore, once placed in a specific directory, the user has no control

over his program. We have found an interesting solution which supports both indirect server referencing and direct user control of server files.

Our first idea was to use hard links to maintain two links to the server. One from the user's directory, and one from a special PARPC server directory. Unfortunately, hard links cannot span file systems. Our solution was to create a special directory on each file system which would contain hard links for user servers within that file system. We then have soft links from a global servers directory into each of the file system specific directories. A PARPC utility called *mkserver* creates the necessary links.

This left one problem unsolved: how to clean up the global and local server directories when the user removes a server file. We chose a straightforward solution. A demon periodically scans the PARPC directories removing any servers which the user has unlinked (i.e. files with one remaining hard link).

With our solution for placing server files, PARPC extends the UNIX development and administration models for distributed programs. Users who develop PARPC programs administer their own files without the support of a system administrator.

4.3. PARPC Administration Tools

The only requirement placed on an administrator is the maintenance of the user equivalence file around the network. However, even this minimal requirement proved to be tedious in the heterogeneous, non-uniform environment at UCSD. To help maintain the equivalence file, we developed two administration tools. The tools are themselves simple PARPC programs. *Reread* executes a parallel procedure at a set of hosts to request that the parent PARPC server reprocess the user equivalence file. This allows network users to be added or removed without having to reboot the servers. *Rgetuid* executes a parallel procedure at a set of hosts to get the local representative of a user and outputs the information in a format suitable for the user equivalence file. Just as the program development tools support distributed program development from a single location in a heterogeneous, non-uniform environment, *reread* and *rgetuid* support administration from a single location.

By extending the UNIX model of access control, resource accounting, program development and administration, our experience has been that PARPC administers itself.

5. Performance Measurements of the PARPC System

The PARPC runtime environment has good performance, especially in light of the fact that it was not implemented in the kernel and that it provides a high level model of computation.

Figure 6 gives timings for null procedure invocation and return using PARPC. PARPC transparently supports three types of procedure calls. *Remote procedure call* is a procedure call made from a process on a local machine to another process on a remote machine. A *local remote procedure call* is made between two processes on the same machine. A *local PARPC procedure call* preserves PARPC semantics and syntax but is made within the same local process. Timings for initial calls are given for both the resident and user server models. Timings for subsequent calls are the same for both server models. Timings for C function calls are also included for reference.

The timings reflect communication overhead between two SUN 3/60 workstations connected by an ethernet.

Subsequent procedure invocations execute in approximately 6 milliseconds. For comparison, Sprite OS designers report that an inter-kernel null procedure invocation and return between two SUN 3/75 workstations takes 2.8 milliseconds. [Oust88] The timings for Sprite only include execution in the kernel. Our figures reflect a round trip between two user processes.

Invoking the first procedure is substantially more expensive than invoking subsequent procedures because of overhead executing TCP/IP connection protocols, host name resolution, PARPC authentication protocols and remote process initiation. The initial call executed at a resident server does not require invoking a remote process; the first procedure is invoked at a

Null procedure invocation and return	Milliseconds	95% C.I.
Subsequent local remote procedure call	5.766	±.008
Subsequent remote procedure calls	6.285	±.008
Initial remote procedure call to resident server	444.22	±12.8
Initial local remote procedure call to resident server	435.47	±16.4
Initial remote procedure call to user server	634.06	±19.4
Initial local remote procedure call to user server	677.26	±20.6
Local PARPC procedure call	0.139	±.001
C function call	0.00537	±.00001

Figure 6: PARPC Communication Overhead Between Two Sun 3/60 Machines (Average of 128 trials)

resident server in approximately 70% of the time it takes to invoke the first procedure at a user server. The startup time is not a problem for interactive programs and long lived programs. For programs that make a lot of parallel procedure calls, the initial cost is amortized over future calls.

6. Conclusions

We have described our experiences in designing and using the PARPC system. A program making parallel procedure calls executes as a *network process* and has good communication performance.

PARPC supports heterogeneous, non-uniform environments by extending the simple, well understood UNIX model for local program development and execution and system administration. Because the system uses existing UNIX mechanisms to extend the model, our experience has been that PARPC is easy to use and administer.

References

[Bern83] Bernstein, P. and Goodman, N. "The Failure and Recovery Problem for Replicated Databases." *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing,* Montreal, 1983. pp 114-122.

[BiNe84] Birrell, A. and Nelson, B. "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems,* Vol. 2, No. 1 (February 1984) pp 39-59.

[BuMaPa87] Burkhard, W. A., Martin, B. E. and Paris, J. F. "The Gemini Replicated File System Testbed." *Proceedings of the Third International Conference on Data Engineering,* Los Angeles, California.

[CaLoPa87] Carroll, J. L., Long, D.D.E, and Paris, J.F., "Block Level Consistency off Replicated Files." *Proceedings of the Seventh International Conference on Distributed Computing Systems,* West Berlin, West Germany.

[ChZw85] Cheriton, D. R. and Zwanepoel, w. "Distributed Process Groups in the V Kernel." *ACM Transactions on Computer Systems,* Vol. 3, No. 2 (May 1985) pp. 77-107.

[Coop83] Cooper, E. "Writing Distributed Programs with Courier." UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Computer Systems Research Group, Computer Science Division, University of California, Berkeley, August, 1983.

[Coop85] Cooper, E. "Replicated Distributed Programs." Ph.D. Thesis. Technical Report UCB/CSD 85/231. University of California, Berkeley, May, 1985.

[Feld79] Feldman, S.I. "Make -- A Program for Maintaining Computer Programs."" UNIX Programmer's Manual, Jan. 1979, Bell Laboratories.

[Giff79] Gifford, D. K. "Weighted Voting for Replicated Data." *Proceedings of the Seventh ACM Symposium on Operating System Principles,* 1979, 150-161.

[John79] Johnson, S.C., "Lint, a C Program Checker." UNIX Programmer's Manual, Bell Laboratories.

[KeRi78] Kernighan, B. W. and Ritchie, D. M. "The C Programming Language." Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Martin86] Martin, B. E. "Parallel Remote Procedure Call: Language Reference and Users' Guide." Technical Report CS-097, UCSD Department of Electrical Engineering and Computer Science, July, 1986.

[Martin87] Martin, B. E. "Modeling Concurrent Activities with Nested Objects." *Proceedings of the Seventh International Conference on Distributed Computing Systems,* West Berlin, West Germany.

[Martin88] Martin, B. E., "Concurrent Nested Object Computations." Ph.D. dissertation, UCSD Department of Computer Science and Engineering. June, 1988.

[MaBeRu87] Martin, B. E., Bergan, C.A., and Russ, Brian, "PARPC: A System for Parallel Procedure Calls" *Proceedings of the 1987 International Conference on Parallel Processing,* The Pennsylvania State University Press.

[MeBo76] Metcalfe, R. and Boggs D. "Ethernet: Distributed packet switching for local computer networks." *Communications of the ACM,* July 1976.

[Oust88] Ousterhout, J., Cherenson, A., Douglis, F., Nelson, M., Welch, B., "The Sprite Network Operating System" *Computer,* February 1988.

[Saty86] Satyanarayanan, M. *RPC2 User Manual.* Rep. CNMU-ITC-84-036, Information Technology Center Carnegie-Mellon U. (July 1986).

[Stro86] Stroustrup, B. "The C++ Programming Language." Addison-Wesley, 1986.

[Sun84a] Sun Microsystems, "External Data Representation Reference Manual." Mountain View, California, October, 1984.

[Sun84b] Sun Microsystems, "Remote Procedure Call Reference Manual." Mountain View, California, October, 1984.

[TaRe85] Tanenbaum, A. S. and R. van Renesse, "Distributed Operating Systems," *ACM Computing Surveys* 17, 4 (Dec. 1985), 419-470.

Dynamically Synchronized Locking - a Lightweight Locking Protocol for Resource Locking in a Stateless Environment

Peter Gloor, Rudolf Marty
Institut für Informatik
University of Zurich
Winterthurerstrasse 190
CH-8057 Zurich,Switzerland
E-mail: gloor@ifi.unizh.ch, gloor@unizh.uucp

Abstract

File and record locking is one of the dominant problems for a stateless file server. By definition, a stateless server does not maintain any information about its clients. Therefore, it is not allowed to lock any resources for them by storing lock information on behalf of its clients. This is the reason why filesystems with locking capabilities are frequently implemented following the stateful approach.

We introduce a new locking method for resource locking in a stateless environment. Our method combines the advantages of the stateless server (easy crash recovery) with the advantages of the stateful server (easy locking) without relinquishing the statelessness of the server. DSL can be used to implement exclusive (e.g. read/write) or shared (e.g. read only) locks. The algorithm we propose (called Dynamically Synchronized Locking, DSL) can be used e.g. to implement locking facilities in a network of workstations loosely coupled by a high speed LAN. It will be exemplified by the implementation of a lock library function for a distributed system which is comparable to the UNIX lockf() System Call [Roc85].

1. Introduction

1.1. Characteristics of the Stateful and the Stateless Client-Server Model

In an environment which is organized following the *client-server model*, we distinguish between two kinds of processes: The client process (the client, for short) requesting some service, and the server process (the server) providing the service demanded by the client.

Moreover, we also distinguish between stateless and stateful servers. The conventional approach is the *stateful server*. A stateful server stores all necessary information about its active clients. To illustrate the stateful approach, we consider a file server: A client of a stateful file server initializes a session with the server by explicitly opening the session. The session will be registered in a table on the server and in a table on the client. The client may then read or write any files from the server essentially as it would do on its own local file

system. When the client wishes to terminate the session with the server, it has to explicitly close the session. Using UNIX terminology, a session has the following form:

```
session_id = open(server,client_authentication,file,...)
  ↓
while (...) {
  read(session_id,&buffer)
  ....
  write(session_id,&buffer)
  ↓
} close(session_id)
```

In the *stateless server* model the server does not store any client information at all. This means that the client has to augment every request to the server with all information the server needs to process this request. There exists no session between client and server. A dialogue with a stateless file server follows this structure:

```
  ↓
while (...) {
  read(server,client_authentication,file,&buffer...)
  ....
  write(server,client_authentication,file,&buffer...)
}
  ↓
```

Clearly, a conversation with a stateless file server entails a considerable overhead because a lot of additional information has to be sent along with every request.

1.2. Drawback of the Stateful Server in a Network Environment

The stateful server model has one big disadvantage which motivates the use of stateless protocols mainly in a networking environment: A stateful server has always to be informed about the state of its clients. This is a minor problem in a single host environment where client and server processes are located on the same machine. A crash will most likely kill both the server and the client. In a networking operating system, where client and server processes may be located on different machines, monitoring the state of all its clients is a real problem for the server. This means that the designer of services following the stateful client-server model has to implement a complicated error recovery protocol to update the state tables of servers and clients if either of them crashes.

In a stateless environment, on the other hand, we need not bother about client or server crashes. Because the server does not store any information about its clients at all, it does

not need to take any special actions after a client crash. Also, the client does not distinguish between a slow but correctly running server and a server which went down and came up again [Ecm85] [San86]. We see that all sorts of error handling and failure recovery problems are much easier to treat in a stateless server environment. In the introduction we mentioned that file and record locking is a dominant problem in a stateless environment because a stateless server is not allowed to store any information about its clients. In order to lock a particular resource for a client the server has to store this client-related information. This means that locking is a stateful service which can not be implemented directly in a stateless environment.

1.3. Existing Approaches to Solve the Stateless Locking Problem

In order to solve the locking problem, any mutual exclusion algorithm can be used. The easiest way would be to choose the central approach which means that the server administrates the global lock table. This results in the stateful server model we want to avoid.

Suns Lock Manager
An example of such a solution is Sun´s NFS [San86]: The locking mechanisms for the stateless filesystem are implemented using an additional server which monitors the state of all network-nodes. Every machine granting or demanding a lock uses this additional supervisor. Sun implemented this supervisor called *State Monitor* [Cha85] as an additional server residing on dedicated sites, these sites being generally more stable than the diskless workstations: The clients are divided in groups. Each client group is monitored by one server. The server probes for the state of its clients by periodically sending them a multicast status message which has to be answered by the clients. The lock manager, a further service also running on every machine, uses the supervisor for keeping its lock tables in a consistent state: After a server crash, a "grace recovery protocol" is used: On detecting a server´s recovery, the client lock manager of a client holding a lock sends a reclaim request. During the "grace period", such reclaim requests are treated in a special manner by the server´s lock manager. This flavour of locking, therefore, is a stateful service, and introduces state into a stateless client-server model, albeit a "distributed version of state".

Theoretical Approaches
A (at least theoretically) better approach should be based on a decentralized algorithm such as a queue based algorithm (e.g. Lamport´s Algorithm [Lam78]) which is particularly well suited for a network with a broadcast facility or a token based algorithm (e.g.[LeL77]) which is convenient for a network with a ring topology. Assume that we will build our locking service in a network with a broadcast facility (e.g. a CSMA/CD network like Ethernet [Tan81]). Thus we could use the Lamport algorithm for the implementation of the locking service. This algorithm requires that we build a queue which will be replicated on every node. Every client requesting a lock broadcasts the lock notification to every other client which places the requests in its FIFO queue. The result is a locking queue replicated on every client. Every client receiving a lock notification immediately sends an acknowledge to the initiating client. A client might get the lock if it is the first in the queue and knows (because of the acknowledgements) that the resource is not yet locked. To unlock a resource a client broadcasts a release message to every client. To schedule the acquisition of a lock all interprocess communication messages are time-stamped. If a client crashes all other clients have to be notified and all requests of this particular client have to

be deleted in all lock queues. Evidently, this algorithm is not well suited for coping with crashes. First, every client needs to know of the existence of every other client. This means that we need some kind of global state information which is hard to get and unstable in case of failures. Second, the recognition of a client crash is not built into the algorithm, but has to be noticed some other way.

2. Basic Structure of the DSL-Protocol

To give a simple introduction into the basic structure of the DSL-protocol, we make the following assumptions (which will be loosened in the later discussion):

- An **infinitely fast network** without any transmission delay
- A **reliable broadcast** channel that **does not require the recipients to be explicitly named.**

Conceptually, there is one logical lock table for the entire network which is physically replicated on every active client. Every client is responsible for the consistency of its own copy. After system initialization time all clients and all servers are up with an empty lock table, i.e. there are no locks. If a client wants to lock a resource, it checks its local copy of the lock table. If there is no entry for this resource, the client makes an entry in its copy of the lock table and broadcasts this so called *lock notification* to all other clients which also make an entry in their lock table. To release a lock, the client deletes the entry locally and broadcasts the delete message to all other clients. If a client´s lock table contains a lock for a resource the client intends to lock, the client either aborts or waits until the lock is released and then continues as above.

In order to make DSL operationally reliable three open problems have to be solved:

- A client crash has to be made known. To solve this problem we will present a method based on timestamps.
- The network delay has to be taken into consideration. This means that there is a finite delay between transmitting a lock request by a client and the registration of this lock by the other clients.
- If the reliability constraint of the broadcast is relinquished, we have to introduce an additional protocol in order to keep the lock tables consistent.

We will address these problems in the following paragraphs.

3. Coping with Client and Server Crashes

In a non-ideal network we have to cope with the possibility of client and server crashes. *Server crashes* need not bother us since the server has no state. But a client crash will likely block some resources and lead to inconsistent lock tables. We distinguish between five classes of client crashes:

*a) A client holding **no** locked resources crashes and comes up again*
A newly initialized client starts with an empty lock table. If it demands a lock for an already locked resource, the client, which has already locked the resource before (the *lock holder*), sends a so called *lock protest* to the new *lock-initiator*, whereupon the new lock-initiator (and all clients, which do not have an entry for this resource in their lock table) will update their lock tables accordingly.

*b) A client holding locked resources crashes and **never comes up again**.*
Because we do not want to use an additional state monitor, we introduce a method based on lock entries augmented with timestamps based on what we call an **approximately synchronized global clock** [1]: Every lock notification is broadcast together with a timestamp. If client A demands a resource which has been locked by client B "a long time ago", it can detect this immediately from its lock table. Client A now broadcasts its *lock notification* for the resource in spite of the fact that the resource already had been locked before. If client B is still up and using the locked resource, it sends a so called *lock protest* to client A. If, on the other hand, client B does not react, client A assumes client B to be down. Consequently client A broadcasts the message "client B is down" to all other clients, which in turn delete all entries for client B in their lock table[2]. We have introduced here the concept of a **decaying lock**. If a lock is held for a period of time by some client (exceeding the decay time), it is assumed to be invalid. If, on the other hand, the lock is still valid, the new lock-initiator has to treat a complaint of the old lockholder. The time for which a lock is definitively valid (the decay time) must be chosen carefully. If the decay time is too short, too many locks which are still valid will be demanded; if it is too long, resources which are demanded by crashed clients will be blocked too long, thus degrading the performance of other clients.

*c) The client holding locked resources crashes and **comes up "after a long time"**.*
"After a long time" means that all old lock entries have been detected by the method described above by the other clients such that there are no old lock entries for this client in the lock tables of the other clients. The client itself comes up with a newly initialized (empty) lock table. If the client initiates a new lock request, the resource is either available or locked by another client. We have already discussed the treatment of this situations since the latter corresponds to claiming a lock after its decay time has elapsed.

*d) The client holding locked resources crashes and **immediately comes up again**.*
We now address the case where a client B comes up (with an empty lock table) while some or all of the locks it held when crashing are still contained in other clients´ lock tables. The problem is that as soon as some other client, say A, has detected the decay of a lock for a resource it wants to lock itself, this client will execute the protocol introduced for crash class b) above. In the simplest case, client B is up again and answers the query "are you up?" of client A positively. But because B comes up with an empty lock table, it does not remember its old locks. Thus, some resources may be locked infinitely. In a more complicated case, client B (which has initiated the decayed lock) is not up yet and does not

---

[1]Exact limits for the approximately synchronized clock will be given later in the paper.

[2]Practically this scheme has to be implemented in two steps: In the first step client A sends an explicit "are you up" message to client B. If client A does not get any answer it broadcasts in the second step the "client B is down" message to all clients.

respond to the lock request from A. A now concludes that B is still down. Let us further assume that B comes up and locks resources before A has broadcast the "client B is down" message. This broadcast would now result in the deletion of all locks of B in all tables, including the ones B has acquired after its crash. To prevent this, we store an additional parameter with every lock: the **boot count** of the client holding the lock. Every time a client is booted, it increments its boot count by one and writes it to stable storage. Whenever a client A assumes that client B is down (because of a decayed lock), it asks client B, whether it is still up and running with the correct boot count. If client A does not get a positive answer, it sends a broadcast message to all other clients to the effect that all locks of client B with this particular boot count are invalid and can be deleted.

e) Multiple client failures
Multiple client failures do not disturb the DSL protocol. Because the clients do not store any state information about the other clients, they may even not notice the failure of other clients, until they need a lock for a resource of a crashed client, whereas they will detect the failure by means of the decayed lock.

4. DSL in a Non-Ideal Network

4.1. Inclusion of the Network-Delay - Colliding Lock Requests

We assume for our considerations that communication between processes running on the same machine is much faster than interprocess communication between remote machines. Thus it is possible that two remote clients send a lock notification at the same time[1]. To solve this problem we suggest a method based on collision detection: After having sent its lock notification, the lock-initiator suspends itself for a certain amount of time. When continuing, it checks whether a lock notification for the same resource arrived from another client. If two lock notifications for the same resource cross each other, both clients wait a random amount of time before initiating their lock request again. Because of this random delay, the probability of the retransmitted lock request to again collide with the retransmitted lock request of the other lock-initiator is minimized. (If it collides, all will begin again.) In this respect the algorithm is identical to a CSMA/CD network [Tan81].

4.2. Unreliable Broadcast Resulting in Inconsistent Lock Tables

In a local area network like Ethernet the broadcast mechanism is not fully reliable: A broadcast message will reach the nodes with high probability, but without an additional protocol, there is no secure information distribution. Consequently we have to relinquish the reliability constraint of the broadcast channel. In other words, we need an additional way of keeping the lock tables consistent. We will propose a distributed method, which

---

[1] Here we have the problem of the "same time". Actions are said to be performed in the "same time", if, at the start of their execution, they are not aware of each other because of the network delay.

does not induce additional network traffic and prophylactic error recognition. Instead, we make use of the fact that broadcasting lock acquisitions enables all clients to permanently maintain their local lock tables. An upcoming client begins with an empty lock table. If it demands a lock for an already locked resource, all other clients on the network and also the client which holds the lock of the resource become aware of this fact. Thus, the lockholder can defend its lock by sending a broadcast message to all other clients (and therefore also to the lock-initiator demanding the already locked resource). Every client which lacks a lock entry for this resource in its lock table adds the lock. Of course this scheme works not only for upcoming clients, but for every lock-initiator which has, by incident, an incorrect lock table.

As said before (4.1), the lock-initiator waits a certain amount of time after sending a lock notification. While waiting it can accept the deletion of its lock request by the owner of an older lock for the same resource. This means that the waiting time has to be at least as long as the time it takes for the lock notification message to reach every client, the optional treatment by another lockholder, and the lock protest message to reach the lock-initiator. If the lock-initiator gets a lock protest after the "successful" termination of the DSL locking protocol, the transaction has to be aborted. Because we want to keep the probability of aborting a transaction as small as possible (preferably zero), we have to choose the delay between sending a lock request and continuing the transaction very carefully. As can be seen above, the time must be at least twice the network delay time between the farthest nodes plus the time needed by the slowest host to trigger a protest message. If this time is too large, the performance of the locking mechanism degrades unacceptably. Surely, the determination of this waiting time is one of the critical points for the correct and efficient working of the algorithm. The solution, as we have proposed it here, does not guarantee correct lock tables at every instant. This means that instead of error prevention or error avoidance we suggest dynamic error correction. Just as with memory management, where the necessary pages are loaded into the main memory only after a page fault occurs, the clients load the correct locks into their lock tables after a lock fault occurs.

As mentioned before every newly initialized client begins with an empty lock table. If an upcoming client first copies the lock table from some other client already up, an evident reduction of initial lock faults can be reached. This improvement does not change the algorithm, but brings a better lock performance for a newly initialized client.

5. Summary of the DSL Protocol

(For a description of an exemplary implementation of DSL under Sun's NFS see [Glo88].)

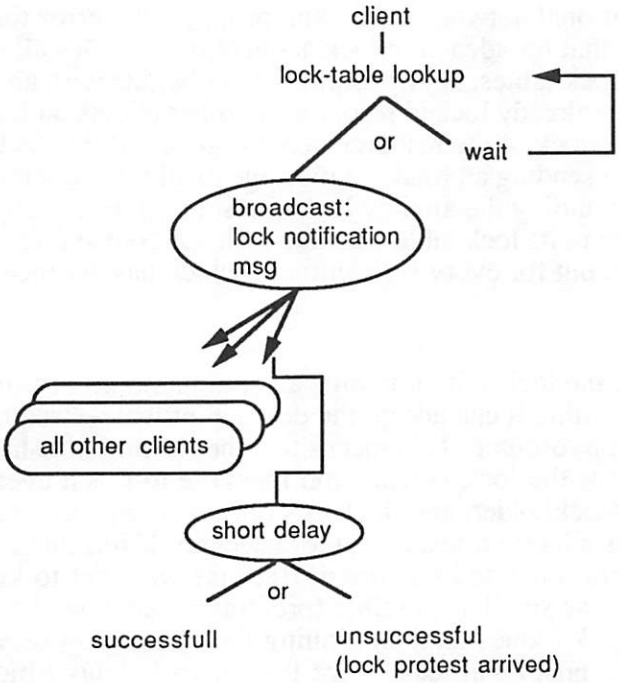


Fig. 1. Execution of a lock request for the stateless server model

Assumptions:

- An *approximately synchronized global clock.*
  (The difference between the clocks has to be smaller than one third of the lock decay time[1].)
- A *reliable broadcast channel,* where a message reaches all nodes at about the same time.

There are two **time constants** which have to be chosen carefully:

- *Networkdelay:* the time a lock-initiator suspends its actions after it has broadcast a lock notification. This waiting time is needed for the detection of lock collisions and for the detection of invalid entries in the lock-initiator's lock table. (Networkdelay >= 2 * longest network delay time + processing time of slowest client)
- *Decaytime:* the time a lock is assumed to be valid by clients not holding the lock.

---

[1]Exactly, the difference between the clocks has to be smaller than half the lock decay time. Otherwise locks will be noticed as decayed as soon as they will be registered in the client lock tables. It is supposed here that the distributed operating system offers a loosely synchronized global clock. There will be taken no additional measures in order to notice a divergence of the local clocks. In the worst case (diverging clocks), at every lock request for an already locked resource the old lock entry is supposed to be decayed and the additional protocol for a decayed lock is executed. But the correct working of the DSL protocol is granted further on.

Description of the locking protocol:

1. The lock tables are maintained by the clients.
2. Every lock request is broadcast to all other clients by the lock-initiator.
3. A lock table entry for a resource consists of the 4-tuple (resource id, client id, timestamp, boot count).
4. For the detection of crashed clients a timestamp based method in combination with the lock table is used. The locks decay, meaning they become "invalid" after a decay time.
5. To detect clients that crashed and came up again quickly, a boot count is used. Upon booting a client the boot count is read from stable storage, increased by one and written to stable storage.
6. For the detection of conflicting lock requests (caused by the network delay), a collision detection method is used.
7. Before the lock-initiator continues its operations, it waits for a period of time given by the constant *Networkdelay* (see above).
8. For the detection of incorrect entries in the lock tables an error correction method is used.
9. The lock table of a newly initialized client can be empty or can be copied from any other client which is up and running.

Definition of the client tasks in pseudocode

```
my_lock:= F;   /* TRUE if the resource is already locked locally */
other_lock:= F; /*TRUE if another lockholder sends a lock protest*/
old_lock:= F;   /* TRUE if another client already holds a lock
                which is older than decaytime for the same
                resource*/
conflict:= F;   /* TRUE if another lock notification for the same
                resource arrives */
```

1. On system initialization time

```
initialize send and receive port to broadcast channel;
initialize lock-table by copying it from any other client which is
already up (or initialize empty table);
get old boot count, increment it by one and write it to stable
storage;
```

2. Locking a resource

```
IF my_lock THEN
    RETURN(already_locked);
IF old_lock THEN
    broadcast(client x with boot count b, please respond if
          up);
    wait(network-delay);
    IF client_response THEN
          RETURN(already_locked);
    ELSE
```

```
            delete in lock-table all locks for client x with boot
                  count <= b;
            broadcast lock-notification;
            wait (network-delay);
            IF ¬conflict ^ ¬other_lock THEN
                  my_lock:= T; RETURN(lock_ok);
            ELSEIF ¬conflict ^ other_lock THEN
                  other_lock:= F; RETURN(already_locked);
            ELSEIF conflict ^ ¬other_lock THEN
                  conflict:= F; wait(randomtime);
WHILE ¬my_lock ^ ¬ other_lock DO
      broadcast lock-notification;
      wait (network-delay);
      IF ¬conflict ^ ¬other_lock THEN
            my_lock:= T; RETURN(lock_ok);
      ELSEIF ¬conflict ^ other_lock THEN
            other_lock:= F; RETURN(already_locked);
      ELSEIF conflict ^ ¬other_lock THEN
            conflict:= F; wait(randomtime);
```

3. On receiving a lock notification message

```
IF my_lock THEN
      broadcast lock_protest;
ELSE
      update lock-table;
```

4. To unlock a resource

```
IF my_lock THEN
      broadcast unlock;
      delete entry;
```

5. On receiving an unlock message

```
IF my_lock THEN
      delete entry;
```

6. On receiving the message "client x with boot count b, please respond if up!"

```
IF I am client x with boot count b THEN
      send to lock-initiator "client x for boot count b is up";
```

7. On receiving the message "client x is down" &"lock notification"

```
delete in lock-table all locks for client x with boot
      count <= b;
IF my_lock THEN
      broadcast lock_protest;
ELSE
      update lock-table;
```

8. On receiving a "lock protest" message (a lock-initiator demanded an already locked resource)

```
update lock-table;
```

6. *Informal Correctness Proof*

It is given here a short correctness proof of the DSL algorithm by showing the correct working of the algorithm under extreme conditions.

- Client A is trying to acquire a lock for a resource which has already been locked by client B, client B is working normally:
  Client B defends its older rights by means of the *lock protest*, (see 3.a and 4.2).
- Client A is trying to acquire a lock for a resource which has already been locked by client B, client B is crashed:
  The lock of client B will be noticed as decayed by means of the concept of the decaying lock (see 3.b).
- Client A is trying to acquire a lock for a resource which has already been locked by client B, client has crashed and is already again up:
  By means of *decaying lock* and *boot count* it will be noticed that the lock held by client B is invalid.
- Client A is trying to acquire a lock for a resource, client B tries to lock the same resource at the same time:
  The network delay is taken into consideration by a waiting time after every lock request. During this waiting time, colliding lock requests for the same resource can be noticed (see 4.1).

7. *Where DSL Might Fail*

DSL is essentially based on a reliable broadcast network. In many environments such as local networks of workstations connected with Ethernet, the broadcast protocol (e.g. UNIX datagram sockets based on the UDP protocol[Tan81]) is reliable enough for not having to implement an additional protocol to ensure reliability. Moreover, DSL is stable enough to work correctly in case of a single broadcast message not reaching every client: Assume that a client did not receive some lock notification and therefore does not know that a resource already has been locked. If it now demands the resource for itself, the client which already had locked the resource before will send a protest message after having received the lock notification message of the new lock-initiator. If this approach is not stable enough we could implement a reliable broadcast protocol as it has been proposed in [Pow83] or [Cha84]. The implementation of a reliable broadcast protocol would entail a significant overhead and add more network traffic. In addition the reliable broadcast protocol demands the exchange of additional messages in case of a broadcast failure. This means that we would have to compute the networkdelay (the waiting time of the lock-initiator) accordingly to the reliable broadcast protocol.

Belated lock protests
There are very rare cases where this algorithm might fail. The problem lies in the determination of the length of the waiting time (the networkdelay). If the lock-initiator has continued its transaction after having waited for an eventual "lock protest" message of another client, a (late) "lock protest" message may arrive. Thus the lock-initiator needs some kind of exception handling facility in order to abort a continued transaction after having waited for the networkdelay time.

Network Partitions
DSL is not stable in case of network partitioning. In every network partition locks for the same resource will be forgiven independently. DSL is therefore better suited for LAN's where an interrupt in the network makes the orderly working of the machines impossibles.

8. Comparison of DSL with other Locking Protocols for Stateless Client-Server Networks

8.1. Comparison with Sun's Lock Manager

We made some tests comparing the speed and the efficiency of DSL with the original Sun locking mechanism. We used different numbers of clients, each client represented by a process residing on different machines. The client processes randomly locked one of ten files and gave it free after a short period of time before trying to lock the next file. The Sun locking mechanism executed our test suite about 60% faster than DSL. This is not surprising considering firstly the implementation of the Sun mechanism inside of the operating system kernel and secondly the waiting time being necessary for every DSL lock request. On the other hand the DSL lock requests were about 70% more successful than the lock requests executed by the Sun mechanism. This shows that for frequently accessed resources the performance of the two mechanisms is comparable. Of course, the execution of a single Sun lock request is about 60% faster than the execution of a single DSL lock request. But the actual Sun implementation entails some considerable disadvantages:

- The Sun lock manager needs an additional state monitor process which has to be informed about the machines it has to monitor.
- The Sun state monitor does not detect a client crash until the client comes up again. This means that some invalid (pending) locks may remain undetected for long periods of time (until the machine holding the pending lock is booted again).
- The Sun lock manager is embedded in the operating system kernel while DSL is running as a normal user process. This means that the complexity of the Sun locking protocol is added to the already relatively complex operating system kernel, while a user level locking service like DSL is much easier to debug and to maintain. But the biggest advantage of our user level DSL-implementation is its portability. By just using plain BSD 4.3 sockets for interprocess communication and running the lock server as a normal user process, it can be ported easily to every UNIX machine supporting the notion of BSD4.3 sockets by just recompiling the source code. (We not only tested our locking server on Sun workstations, but also ran it on a Sony workstation and a Vaxstation.)

8.2. Comparison with Other (More Theoretical) Approaches

The main challenge of DSL is keeping the distributed copies of the lock table consistent. Various algorithms for the detection of inconsistencies among distributed replicas of files are known [Bre83],[Par83]. But these algorithms are not particularly well suited to the stateless locking problem because the absolute consistency of the copies is paid for with a reduction of the availability of the replicated file, i.e. if one inconsistent entry is detected,

all copies will be completely updated. During this update, the lock-table is not available for the locking protocol. The resulting performance degradation is not tolerable for the locking protocol. As Agrawal writes in [Agr85] an optimistic algorithm (like DSL) is a better choice than a blocking algorithm (like the above cited algorithms) for an environment like a local network of workstations where resource utilization is sufficiently low. An interesting approach to detect state inconsistency in local area network based distributed kernels has been suggested by [Rav86]. It demands the creation of an ancillary process for every process which has to be supervised. But failures have to be detected by polling of the ancillary process which results in an additional network traffic and in an additional complication of the locking protocol.

The main advantage of DSL is that there is no global client state information to be stored anywhere. Almost all other synchronization algorithms count on every client to know the total number of active clients. In DSL the clients store only information about locked resources. In the DSL protocol, the client will, with the exception of (the rare case of) colliding lock requests, not demand locking of an already locked resource because it has its local copy of the lock table. Thus, a lock request for an already locked resource

- will return the "already locked" message *without any network delay* and
- will *not cause any network traffic* and

results in a better *efficiency* and overall performance of the whole system. DSL is therefore well suited for a system with resources which are frequently locked. DSL is stable in case of client and server crashes. *Dynamic recovery* in case of a client crash is built into the algorithm and just results in a (temporary) degradation of performance until the client´s lock table is consistent again. All approaches mentioned above either demand some kind of global state information which is hard to maintain in a consistent state or they are inefficient. We therefore claim that our heuristic algorithm which

- renounces on global state information and only uses some kind of local state information and
- offers an efficient implementation with minimal additional network traffic

is well suited for the stateless locking problem.

9. Further Research

We have to distinguish between two possible failure types: network failures and machine (and process) failures. Machine and process failures will be treated correctly by DSL. In case of a network failure we have to ensure the correct progress of the algorithm either by additionally implementing a reliable broadcast protocol or by implementing some additional security mechanisms on the distributed operating system level. We therefore are working to improve the reliability of the protocol in order to get a protocol which is able to work on a less reliable broadcast network. We will just sketch here such an additional reliability service: After receiving a lock notification all clients getting the notification are prepared to send an acknowledge message back to the lock-initiator. In order to avoid a traffic jam on the network, the clients wait some (short) random time with the acknowledgment. In fact, only the first message (of the fastest client) will be sent really. Because there is only one

global broadcast channel, all other clients learn of this acknowledgement and do not need to acknowledge the receipt of the lock notification themselves.

Acknowledgements

We would like to thank K.J. Schulz and Beverly Sanders for help in clarifying our ideas and for the reading of preliminary versions of this paper. We further thank to Martin Zellweger who implemented the main parts of the DSL protocol.

Literature

[Agr85] Agrawal, R.; Carey, M. J.; Livny,M.; "Models for Studying Concurrency Control Performance: Alternatives and Implications", *Proceedings of the ACM-SIGMOD 1985 International Conference on Management of Data*, Austin, Texas (1985)

[Ber81] Bernstein, P.A.; Goodman, N.; "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys*, Vol. 13, No. 2, June (1981)

[Bre83] Brereton, P.; "Detection and Resolution of Inconsistencies among Distributed Replicates of Files", *ACM Operating Systems Review*, Vol. 17, No. 1, (1983)

[Cha84] Chang, J.-M.; Maxemchuk, N.F.; "Reliable Broadcast Protocols"; *ACM Transactions on Computer Systems*; Vol.2, No. 3, August (1984)

[Cha85] Chang, J.; "SunNet", *Usenix Conference Proceedings*, Portland (1985)

[Chu86] Chu, W.W.; Jung, M.A.; "Fault Tolerant Locking FTL for Tightly Coupled Systems"; *Proceedings of the fifth Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, (1986)

[Ecm85] ECMA (European Computer Manufacturers Association); "*Remote Operations - Concepts, Notation and Connection-oriented Mappings*", TR31, December (1985)

[Gar85] Garcia-Molina, H.; Barbara, D.; "How to Assign Votes in a Distributed System", *Journal ACM*, Vol. 32, No. 4; October (1985)

[Glo88] Gloor, P.; Marty,R.; Zellweger,M.,"Implementation of a Locking Protocol for Resource Locking in a Stateless Environment"; *Proc. EUUG Autumn Conference*, Cascais, Portugal, (1988)

[Lam78] Lamport, L.; "Time, Clocks and the Ordering of Events in a Distributed System", *Comm. ACM*, July (1978)

[LeL77] LeLann, G.; "Distributed Systems - Towards a Formal Approach", *Information Processing 77*, B.Gilchrist, (ed.) North-Holland, (1977)

[Par83] Parker, D.S. et al.; "Detection of Mutual Inconsistency in Distributed Systems", *IEEE Transactions on Software Engineering*, Vol. 9, No. 3, May (1983)

[Pow83] Powell, M.L.; Presotto, D.L.; "Publishing: A Reliable Broadcast Communication Mechanism"; *ACM Operating Systems Review*, Vol. 17, No. 5, (1983)

[Rav86] Ravindran. K., Chanson, S.T.;"State Inconsistency Issues in Local Area Network-Based Distributed Kernels", *Proceedings of the fifth Symposium on Reliability in Distributed Software and Database Systems,* Los Angeles, (1986)

[Roc85] Rochkind, E.B.; *"Advanced UNIX Programming"*; Prentice-Hall, Englewood Cliffs, N.J, (1985)

[San86] Sandberg, R.; Goldberg, D.; Kleiman, S.; Walsh, D.; Lyon, B.; "Design and Implementation of the Sun Network File System", Sun Microsystems, Inc., Mountain View, CA (1986)

[Tan81] Tanenbaum, A.; "*Computer Networks*", Prentice Hall, Englewood Cliffs, N.J., (1981)

Trademarks

UNIX is a registered Trademark of AT&T

Integrating UNIX Terminal Services into a Distributed Operating System

Geoffrey M. Lee

Lawrence Livermore National Laboratory
Livermore, California 94550

Abstract

One solution to the problem of integrating the terminal services of a UNIX host into a distributed operating system is to implement the distributed operating system as a guest layer in UNIX and then to add a software interface that runs on the host and translates between the terminal services of the host and the distributed operating system. Such an interface was implemented at the Lawrence Livermore National Laboratory. Design issues and decisions that preceded software development are discussed, followed by details of software operation. Development experiences that would be applicable to a similar effort are described.

Introduction

The integration of host computers into a distributed operating system requires that either a native distributed operating system be implemented for them[1] or software be developed to allow the distributed operating system to serve as a "guest" on a vendor's system.[2] In the latter case, the terminal services of the vendor's system can be made accessible from within the distributed operating system in various ways. For example, a separate gateway computer can be used to translate between vendor network terminal services and the distributed operating system.[3] Alternatively, software that runs on the host computer can do the translation. The Lawrence Livermore National Laboratory (LLNL) provided terminal services from its distributed operating system into UNIX based on the latter approach.

Livermore Interactive Network Communications System

The Livermore Interactive Network Communications System (LINCS) is an object-oriented message-based distributed operating system that provides an environment in which a user interacts with distributed system objects (e.g., files, directories, user contexts, accounts) rather than with individual computers in the network.[2, 4, 5] LINCS supports a variety of computers, including Crays, VAXes, and SUN workstations. On some of the computers LINCS is the native operating system; on others, it is a guest layer imbedded within a commercial operating system (e.g., UNIX).

In LINCS, objects are managed by processes called *servers*. *Client* processes access objects or services by contacting servers. LINCS objects are identified by 32-byte coded records called *capabilities*. Presentation of an

object's capability to the process managing that object is proof of the presenter's right to access it. Because they indicate the extent of access, capabilities include a cryptographic checksum to prevent tampering. While capabilities typically have low mnemonic value, a label, known as a *link name*, can be associated with a capability through a directory server. A finite sequence of link names is called a *chain name*, and it designates an object capability similar to the way a UNIX path name designates a file. Within LINCS, just as in UNIX, each user ID has a corresponding root directory, and the user of a particular network ID can access exactly those objects whose capabilities can be named with a chain name starting at the associated root.

LINCS contains layers of communication protocols that can be generally described using the seven layers of the ISO Reference Model[6] (application, presentation, session, transport, network, link, and physical layers). However, because LINCS was designed to emphasize efficiency over generality, it does not strictly follow the ISO model. Processes communicate by sending a *monolog* (a sequence of related messages) across a stream. The stream is unidirectional and is uniquely identified by a triple composed of the sender's network address, the receiver's network address, and a stream number specifically chosen to distinguish that stream from any others that exist between the two processes.

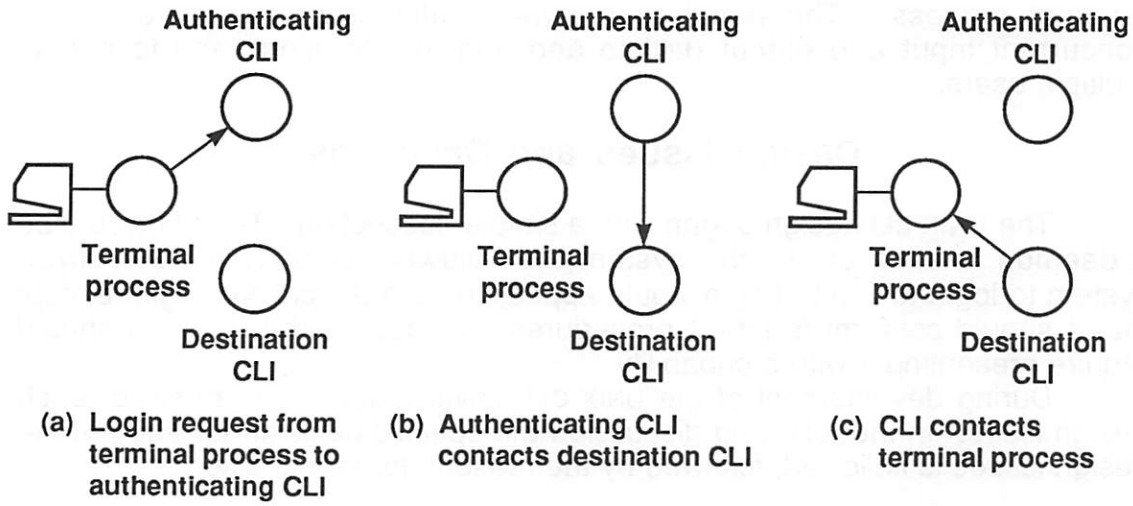
Interprocess communication in LINCS follows the client/server model. Typically, a client sends a server a control monolog containing a series of requests. Each encoded request contains relevant parameters as a series of tokens (presentation elements of LINCS similar to ASN.1 elements), each indicating the type, usage, and value of the parameter. The server responds with a reply that is in some cases another control monolog and in other cases a data monolog (an uninterpreted sequence of data bits). For example, the reply to a request to create a file would be a control monolog containing a token with a capability to that file. A reply to a request to attach to a user context would be a data monolog containing a login banner.

Examples of processes in LINCS are the terminal process that operates the terminal through an I/O interface, an authenticator process that verifies user IDs and passwords, applications that perform computations requested by a user, and command language interpreters (CLIs) that act as intermediaries between the applications and the terminal process. All of these processes may be on the same computer or they may be on several computers.

Terminal Sessions

The steps necessary to establish a terminal session are shown in Fig. 1. The user logs in at a terminal. The login is sent to an authenticating CLI, which authenticates the user before contacting the destination CLI. The destination CLI then establishes a connection with the terminal process. Because of the circuitous flow of information, this procedure is called an "around-the-horn" protocol. Note that validating users on a separate system avoids some of the security problems associated with UNIX logins.

The terminal process and destination CLI use the LINCS virtual terminal protocol (VTP)[4] during the data transfer portion of the session. The connection between the selected CLI and the terminal process can be terminated in one



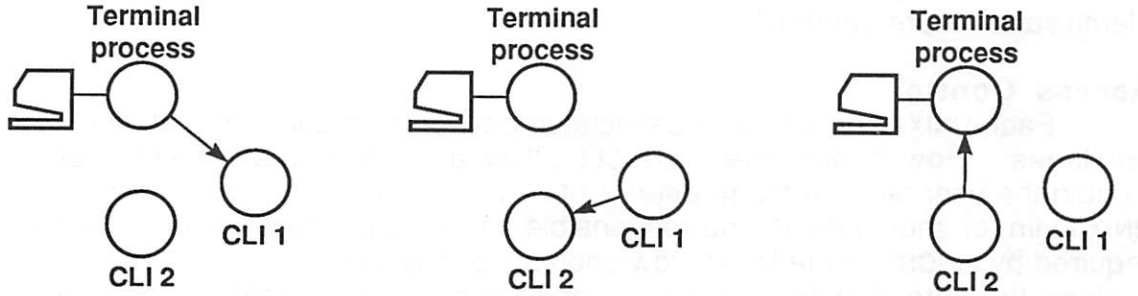
(a) Login request from terminal process to authenticating CLI

(b) Authenticating CLI contacts destination CLI

(c) CLI contacts terminal process

Fig. 1. Establishing a terminal session.

of two ways. The selected CLI can switch the connection to another CLI (Fig. 2). This switch takes place in the same manner that the authenticating CLI switched the terminal connection to the selected CLI. By switching to a different CLI, reauthentication of the user is not required. Alternatively, the terminal can be disconnected from the CLI and a new connection, complete with authentication, can be initiated.



(a) User requests switch to CLI 2

(b) CLI 1 sends switch request to CLI 2

(c) CLI contacts terminal process to establish new session

Fig. 2. Switching from one CLI to another.

CLI Template

Some of the computers in the LINCS network use locally designed and implemented operating systems, while others use commercial systems. One way of interfacing the commercial systems to the network has been to add a guest layer of LINCS modules to them. To facilitate adding LINCS CLIs, a portable LINCS CLI template was created. It contains all of the logic necessary to receive requests, send replies, and maintain interaction with a LINCS

terminal process. The template features multitasking to provide users concurrent input and output (I/O) to and from the terminal, and to handle multiple users.

Design Issues and Decisions

The UNIX CLI design began with a simple description: The CLI would be a daemon running on a UNIX system that allowed users within the LINCS system to login to UNIX. Login would appear to be a direct UNIX login, except that it should conform to LINCS procedures. To access the UNIX CLI should require presenting it with a capability.

During development of the UNIX CLI, several questions arose for each design issue. In the following discussion the specific decision for each of the design issues is italicized, followed by the reasons for that choice.

User Identification

Each user has an identification (ID) within LINCS. That user may also have one or more IDs on the UNIX system being integrated into LINCS. How should the user be identified to the UNIX CLI? Should the UNIX identification be used, or should the LINCS ID be used?

A UNIX ID number in the CLI capability identifies the user.

Sufficient general information is contained in a capability to identify the host system of the CLI. To include a distributed operating system user ID within the CLI would require mapping the distributed system ID into a UNIX ID that is valid on the specific system. We considered this to be an unwarranted complication. The capability includes a UNIX ID number, which makes identification more compact.

Access Control

Each UNIX user ID has an associated environment consisting of files and privileges. How should the UNIX CLI allow access to that environment? Should the user be required to enter a UNIX ID and password, as in the normal UNIX login, or should the CLI be responsible for any authentication beyond that required by LINCS? If the latter, how should the login be done? Should the CLI perform the normal login for the user, or should the CLI execute its own login routine?

The CLI has UNIX super-user status and services its own logins.

For reasons of system security, the CLI should possess only enough privilege to complete its job within a UNIX system, i.e., it should conform to the principle of least privilege.[7] One way to achieve this is for the CLI to log in as the user, thus requiring no more privilege than the user possesses. To do this, the CLI must pass the user's name and password to the normal UNIX login routine. This requires that the user's unencrypted password be available to the CLI and that the password be automatically updated whenever it is changed on the UNIX system.

These requirements make the approach cumbersome as well as risky. For example, if the password is kept within the capability, it cannot be updated when it changes on the UNIX system. If it is stored in a table kept by the CLI, there is too great a risk that it will be exposed.

If, instead, the CLI has super-user privilege, it can log in a user on its own. The CLI can choose to ignore normal UNIX password verification, relying instead upon user authentication within the distributed system. With proper caution during coding, the extension of privilege is less risky than password storage. Of course, this is not an ideal approach because of the well-known problems with the super-user concept.[8]

CLI/UNIX Shell Communications Channel

The UNIX CLI handles multiple terminal sessions simultaneously. Because the CLI is a single process and sessions must remain distinct, it must create a process per terminal session. How should the CLI process communicate with each session process?

The CLI and user shell communicate through a pseudo-tty.

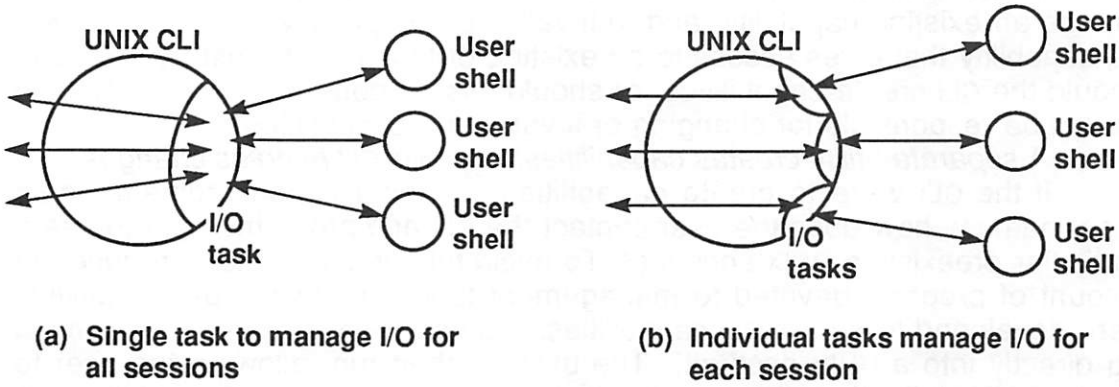
For reasons of portability, pipes were initially preferred to pseudo-ttys. Pipes are available in both UNIX 4.2BSD and UNIX System V; pseudo-ttys are available in 4.2BSD only. However, UNIX systems on different machines treat pipes differently. In particular, the ioctl system calls needed to set terminal characteristics for the communications channel can be applied to pipes on a SUN workstation running UNIX 4.2BSD, while the VAX version of 4.2BSD does not allow all of the necessary calls.

Although pseudo-ttys were not generally available in System V, it seemed easier to implement them on System V than to rework pipes on 4.2BSD. As a bonus, the use of pseudo-ttys mirrored part of the UNIX rlogin daemon logic, thereby reducing new software development.

Separation of Terminal Sessions

Terminal processes throughout the LINCS system send messages to the UNIX CLI, and the CLI relays each message to the proper user process. How should the CLI relay the messages? Should a single CLI task (i.e., light-weight process) manage all terminal sessions by noting the origin of each message, then looking up the corresponding user process (Fig. 3a), or should there be separate tasks to manage each terminal session (Fig. 3b)?

Data messages for a terminal session are handled by a task assigned only to that session, not by a task common to all sessions.



(a) Single task to manage I/O for all sessions

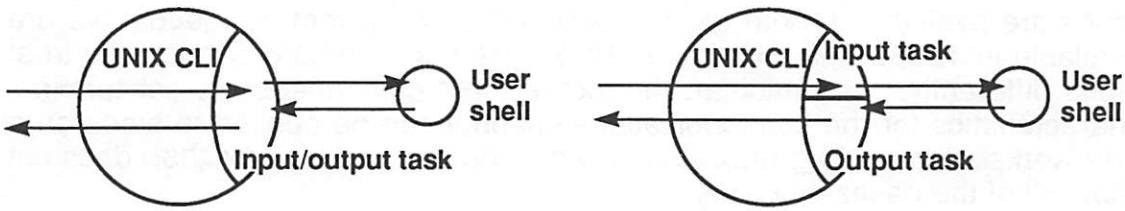
(b) Individual tasks manage I/O for each session

Fig. 3. Managing I/O for sessions.

The philosophy of separate tasks for separate sessions simplifies the logical organization and allows the front end, which handles messages from the distributed operating system, to be written independent of the back end, which handles data from UNIX. The additional overhead incurred in creating an additional task per session is insignificant. This strategy permitted the use of the LINCS CLI template software, which also chose this strategy.

Synchronization of Data

Data messages travel either from the terminal process to the user shell or from the user shell to the terminal process. Should a single task handle both input and output for a terminal session (Fig. 4a), or should there be separate tasks to handle input and output (Fig. 4b)?



(a) Single task to manage input and output

(b) Separate tasks for input and output

Fig. 4. Managing I/O for each session.

One task handles data from the terminal process to the user shell; a separate task handles data from the shell to the terminal process.

A single task handling data in both directions could block data in one direction while it waited for data from the other direction. Again, the additional overhead needed to create two tasks per session is minimal, and such is the choice already made by the template software.

CLI Management

There must be ways to create a UNIX CLI capability for a new user, to change an existing capability, and to invalidate a capability. Creating a UNIX CLI capability that gives access to an existing UNIX account must be possible. Should the CLI create capabilities, or should this be done elsewhere? Should the CLI be responsible for changing or invalidating capabilities?

A separate utility creates capabilities; the UNIX CLI handles changes.

If the CLI were to create capabilities, an initialization problem would arise, namely, how does the user contact the CLI and prove the right to use a particular preexisting UNIX account? To avoid this problem and to reduce the amount of program devoted to management functions, two separate utilities were developed to create CLI capabilities. To use the first utility, a user must log directly into a UNIX account. The utility is then run, allowing the user to create a capability that gives access to that account. The second utility runs as part of the distributed operating system. It allows a system administrator,

using only the distributed operating system terminal services, to create a capability and give it to the user. No other capability management functions have initialization problems; they can be handled directly by the CLI.

Minimization of New Software Development

Duplication of coding effort can be reduced by using existing software whenever possible. What existing software can be applied to the CLI, within both LINCS and UNIX?

Logic was included from the LINCS CLI template, the UNIX standard login routine, and the UNIX rlogin daemon from 4.2BSD.

All of the CLI template was used. The front end provides specific points of interface with the back end, making it easy to tailor a back end to the UNIX system.

Most of the standard UNIX login routine was used. A few code sections unrelated to CLI activity, such as password checking and remote login handling, were omitted; other sections, such as user environment lookup, were modified to fit the situation.

In many respects the CLI is a LINCS version of the rlogin daemon. Parts of the UNIX rlogin daemon used in the CLI handle a variety of duties: selecting an unused pseudo-tty, testing for readiness of a pseudo-tty to accept data, creating a child process to run the user's shell, connecting the CLI process to the child process through a pseudo-tty, and cleaning up during logout.

Portability

The CLI will reside on a diverse set of machines running different versions of the UNIX operating system. How can the CLI be written to maximize its portability from one system to the next?

Maximizing code portability affected the CLI design in several ways. Pseudo-ttys were chosen over pipes because it appeared easier to move an implementation of pseudo-ttys from 4.2BSD to System V than to expand pipe abilities in 4.2BSD.

Although the CLI was developed on a 4.2BSD UNIX system, general C library routines were used whenever possible. Often, this meant overlooking a 4.2BSD system call that performed the function needed, and instead, using a general call that required additional work to achieve the desired goal.

The LINCS distributed operating system infrastructure that runs on UNIX systems includes the support of standard distributed operating system services. Primitives in the interface to this software allow task control and management, memory management, and message services.[9, 10] The ubiquity of this environment makes the LINCS front-end portion of the CLI very portable.

General Operation

The UNIX CLI is divided into two parts (Fig. 5). The front-end logic handles control and LINCS VTP messages from other processes within the distributed operating system; it is the CLI template. The back-end logic handles communications to and from UNIX.

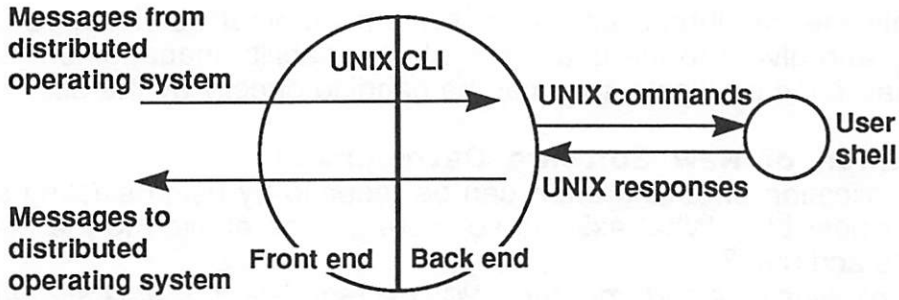


Fig. 5. UNIX CLI front-end and back-end logic.

A LINCS VTP message from a terminal process within LINCS is received by the front end, which performs any necessary steps of the LINCS protocol and then passes the buffer of received characters to the back end. The back end sends the data within the buffer to the UNIX operating system. Responses from UNIX reverse the path back to the terminal process.

From the UNIX point of view, the UNIX CLI is a daemon that handles multiple terminal sessions between LINCS and UNIX and also waits for additional login requests.

Login requests from the LINCS distributed operating system contain a capability. When a request is received, the CLI verifies that the capability has not been altered, then extracts the UNIX ID number from it and creates a new process to run that user's chosen shell.

To keep each session distinct, the UNIX CLI creates two separate tasks per session. One task handles data from the remote terminal and the other handles data from the user's UNIX shell. Data is conveyed through a pseudo-tty between the CLI and the shell process.

When a user concludes a terminal session, the CLI assures that the shell is terminated, the terminal session information within the CLI is discarded, and all associated UNIX resources are freed.

Basic Functions of the UNIX CLI Interface to UNIX

The CLI interface to UNIX performs four basic functions. It opens a UNIX terminal session, transfers data from the remote terminal to UNIX, transfers data from UNIX to the terminal, and closes a UNIX session.

Transferring data from UNIX to the remote terminal is basically the reverse of transferring data from the terminal to UNIX, so in this discussion both directions of data transfer are described in a single subsection.

Opening a UNIX Terminal Session.

When a login request is received, the CLI verifies the capability it contains by recalculating a cryptographic checksum carried within the capability. The user's ID number is extracted from the capability and used to index into a UNIX system file containing encryption keys. The key corresponding to that ID number is then used to compute a checksum on the data portion of the capability, and the result is compared to the checksum

carried within the capability. If the two match, the capability is valid. If the capability is invalid, the login fails.

Given a valid capability, the CLI sets up a separate process to run the user's chosen shell (Fig. 6). An unused pseudo-tty is acquired and a child process is created. The parent process records the number of the child process and the file descriptor of the pseudo-tty for future reference. The child process duplicates the pseudo-tty as its standard input, output, and error output, then executes an abridged version of the standard UNIX login. The abridged login does not verify the password; it accepts possession of a CLI capability as evidence of authentication. The login procedure changes the effective user ID of the child process to the user ID contained in the capability.

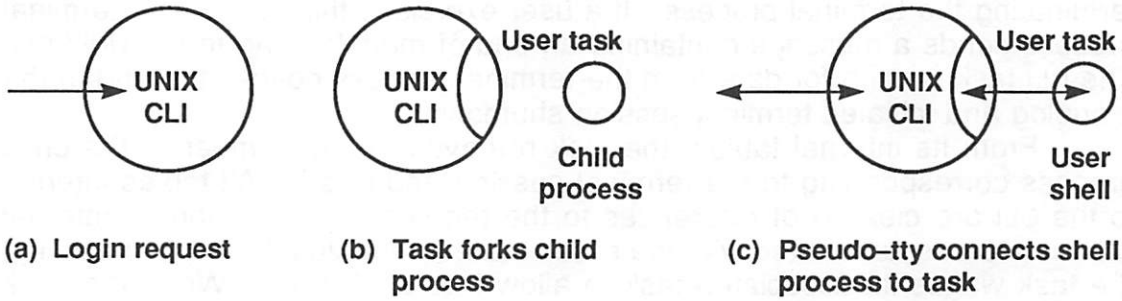


Fig. 6. Opening a terminal session.

Transferring Data Between the Remote Terminal and UNIX

The CLI front end provides two tasks per terminal session (Fig. 7). One task waits for data from the terminal process. When data arrives, that task determines if the pseudo-tty to the user's shell can accept data. If so, the data is written to the pseudo-tty, which relays it to the user's shell. Buffer size constraints may require the task to write to the pseudo-tty several times before all of the data from the terminal process has been transferred.

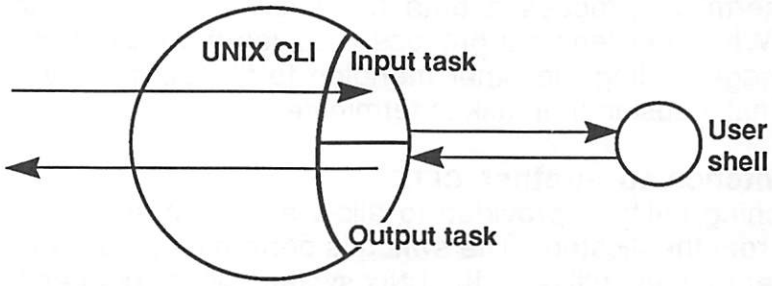


Fig. 7. Data transfer for single session.

Independently, another task checks for the presence of data in the pseudo-tty from the user's shell. The check is performed without blocking the CLI process, thus allowing the CLI to perform other duties while the task waits for data. When data from the shell is detected, the task reads the data from the pseudo-tty and passes it to the terminal process. As with its associated

task, this task may read from the pseudo-tty several times before all data from the shell has been transferred.

Closing a UNIX Terminal Session

Logout may occur in one of three ways: the user may end the session with the terminal process, the user may perform the normal UNIX logout, or the user may switch to another CLI. Each case requires separate logic within the CLI, but in all cases the user's shell is terminated, and any information associated with the terminal session within the CLI is discarded.

The user ends the session with the terminal process

For security reasons, a user must have an unequivocal way of terminating the terminal process. If a user exercises this option, the terminal process sends a message containing an end-of-monolog flag to the UNIX CLI. The CLI task waiting for data from the terminal process notices the end of the monolog and initiates terminal session shutdown.

From its internal tables, the task retrieves the ID number of the child process corresponding to the terminal session and kills it. All tables internal to the CLI are cleared of references to the terminal session, and all internal UNIX resources associated with the session are freed. Just before terminating, the task wakes its associated task to allow it to shut down. When that task checks for data from the user's shell, it finds that the pseudo-tty has been released, so it shuts its part of the session down, and terminates.

The user performs a normal UNIX logout

A normal UNIX logout kills the process that was running the user's shell. In response to a signal that a child has died, the CLI determines the process ID of the child. If the process ID matches one of the IDs associated with some active terminal session, then the CLI initiates terminal shutdown. As before, internal CLI tables are cleared, and internal UNIX resources are freed. The task waiting for data from the user's shell finds that the pseudo-tty has been released; it shuts down its part of the task by waking the associated task, sending the terminal process a data message ending the monolog, then terminating. When the terminal process receives the end of the monolog, it sends a message ending the other monolog to the CLI task waiting for data from the terminal, causing that task to terminate.

The user switches to another CLI

A switching utility is provided to allow a user to switch to another CLI without being reauthenticated. The switch is performed as shown in Fig. 8.

The user runs the utility on the UNIX system, which queries the user for a chain name designating the new CLI. Then the switching utility obtains a LINCS local root directory capability from a predetermined UNIX system file belonging to the user. This capability is used to parse the chain name to obtain a capability to the new CLI. The capability is then placed into a temporary file where the UNIX CLI can find it. Finally, the switching utility logs the user out of UNIX. This kills the process running the user's shell.

As in the previous case, the UNIX CLI notices that the user's shell process has died, shutting down the terminal session. During shutdown the

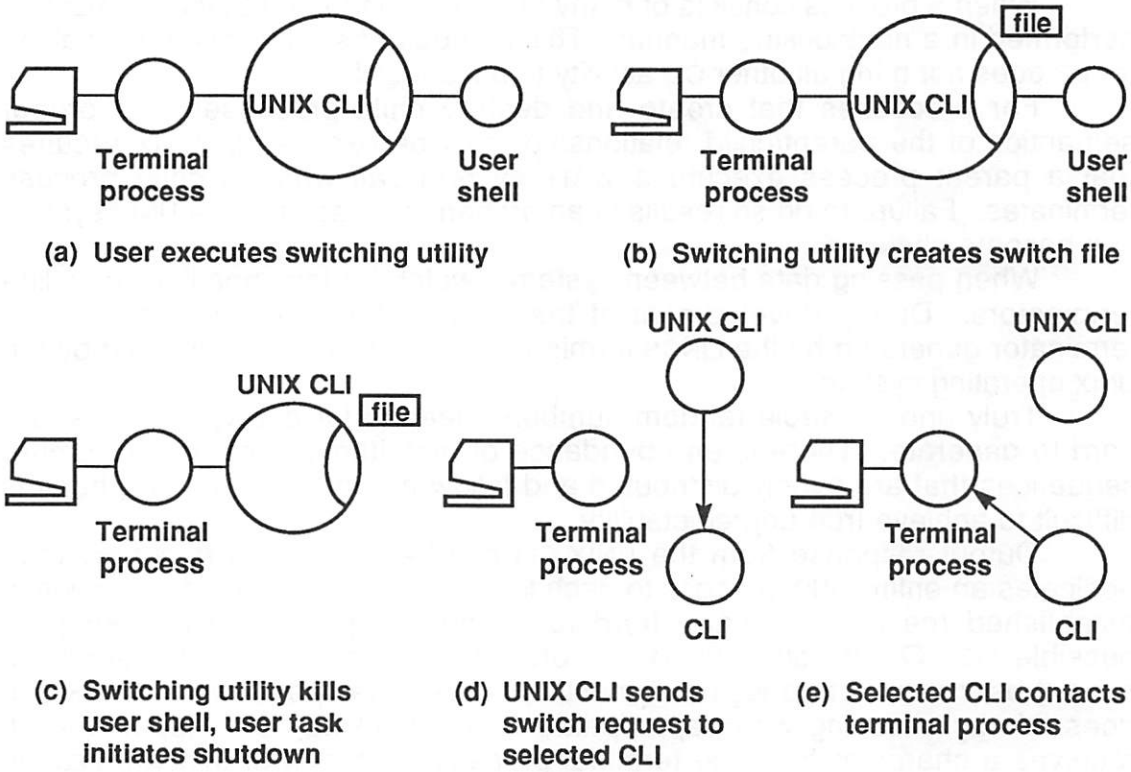


Fig. 8. Switching to another CLI.

CLI checks for the temporary file created by the switch utility, indicating that the user intended to switch to another CLI. If the file is found, the UNIX CLI extracts the capability, prepares a LINCS switch request with the capability of the new CLI, and destroys the temporary file. The UNIX CLI sends the switch request to the new destination CLI, logging the user into the new CLI.

Lessons Learned During Development

We provide, in no particular order, information that could be useful in developing a similar program for other distributed operating systems. While some of the ideas are well known, it is worthwhile to keep them in mind.

Daemon-like processes need to be set up carefully if they are to run (and continue to run) correctly.[11] Most of the setup needed for the CLI was included in the logic from the normal UNIX login routine.

Out-of-band signaling is an important tool in implementing the CLI; unfortunately, such signaling between UNIX processes is not straightforward. UNIX signals are poor out-of-band indicators because they carry so little information. The CLI was unable to use signals alone to indicate that a user wanted to switch to another CLI. Instead, the existence of a special temporary file and a signal indicating the termination of a child process were used to signify that a switch was imminent.

When a process consists of many tasks waiting to do I/O, the I/O must be performed in a nonblocking manner. This is required so that one task waiting for I/O does not bring all other CLI activity to a standstill.

For processes that create and destroy child processes, the proper semantics of the parent/child relationship must be exercised. UNIX requires that a parent process execute a WAIT system call when a child process terminates. Failure to do so results in an orphan process, and the UNIX system can become cluttered.

When passing data between systems, watch for incompatibilities in line terminators. During development of the CLI it was discovered that the line terminator generated by the LINCS terminal process was not recognized by the UNIX operating system.

Truly unpredictable random numbers, needed for encryption keys, are hard to generate. There is an abundance of algorithms to generate number sequences that are evenly distributed and follow no apparent pattern, but it is difficult to achieve true unpredictability.

Output response from the UNIX CLI can be slow. The rlogin daemon dedicates an entire UNIX process to each login, so once the terminal session is established the process can afford to spend its spare time checking for possible I/O. On the other hand, the UNIX CLI is a single process servicing current users and future logins; it must minimize time spent on I/O checks. It does this by checking when appropriate (e.g., checking host output when it receives a character from the terminal process), but lack of synchronization can lead to checks that rely on time-outs, and result in perceivable delays in response.

Related Work

Guest layering is a common way of building a distributed system from existing host operating systems. Several efforts use this approach. The Eden system prototype is layered on UNIX 4.2BSD,[12] with the kernel implemented as a user process. The global distributed operating system[13] adds an Amoeba transaction layer on UNIX 4.2BSD. The Cronus system is layered on UNIX 4.2BSD and VMS.[14] The Desperanto project focuses on research into heterogeneous distributed operating systems with guest layering as a design goal. The system is layered on a VAX running UNIX and on a DEC (Digital Equipment Corporation) System-20 running TOPS-20.[15]

In contrast to the way the UNIX CLI adds an underlying host's UNIX terminal services to LINCS, distributed UNIX systems, such as the Newcastle connection,[16] DUNIX,[17] and the LOCUS system,[18] connect a network of hosts running UNIX so that a user views the entire distributed system as a single UNIX system.

An analogous class of interfaces in which users in the guest distributed operating system access the host system's files rather than user contexts is represented by the LINCS guest file server,[19] which allows LINCS processes to access files in the LTSS (Livermore Time Sharing System) operating system, and the DIOS distributed operating system,[20] in which local file systems are united through a distributed file system.

Acknowledgments

The author wishes to express his appreciation to Dan Nessett and John Fletcher for their help and encouragement in the preparation of this paper. This work was supported by the Office of Safeguards and Security, U.S. Department of Energy, by Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.

References

1. Donnelley, J. E., "Components of a Network Operating System," *Computer Networks*, **3**(6), 389–399 (1979).
2. Watson, R. W., and J. G. Fletcher, "An Architecture for Support of Network Operating System Services," *Computer Networks*, **4**(1), 33–49 (1980).
3. Nessett, D. M., *The Inter-Authentication-Domain (IAD) Logon Protocol*, Lawrence Livermore National Laboratory, Livermore, Calif., UCID-30207 (Rev. 2) (1988).
4. Fletcher, J. G., *Distributed Terminal Interaction Using LINCS*, Lawrence Livermore National Laboratory, Livermore, Calif., UCRL-98839 (1989).
5. Watson, R. W., *Requirements and Overview of the LINCS Distributed Operating System Architecture*, Lawrence Livermore National Laboratory, Livermore, Calif., UCRL-90906 (1984).
6. ISO, "Data Processing—Open Systems Interconnection—Basic Reference Model," *Computer Networks*, **5**(4), 81–118 (1981).
7. Saltzer, J. H., and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, **63**(9), 1278–1308 (1975).
8. Ritchie, D. M., "On the Security of UNIX," *UNIX System Manager's Manual*, (4.2 Berkeley Software Distribution, 1984).
9. Fletcher, J. G., "APST Interfaces in LINCS," Lawrence Livermore National Laboratory, Livermore, Calif., working document, July 1985.
10. Fletcher, J. G., "SMILE," Lawrence Livermore National Laboratory, Livermore, Calif., working document, April 1988.
11. Lennert, D., "How to Write a UNIX Daemon," *;login: The USENIX Association Newsletter*, **12**(4), 17–23 (1987).
12. Black, A., "Supporting Distributed Applications: Experience with Eden," Proc. 10th ACM Symp. on Operating System Principles, Orcas Island, Wash., December 1–4, 1985, *Operating Sys. Rev.*, **19**(5), 181–193 (1985).
13. Turnbull, M., "Support for Heterogeneity in the Global Distributed Operating System," *Operating Sys. Rev.*, **21**(2), 11–21 (1987).
14. Schantz, R., R. Thomas, and G. Bono, "The Architecture of the Cronus Operating System," *IEEE Proc. 6th Internl. Conf. on Distributed Computing Systems*, Cambridge, Mass., May 1986 (The Institute of Electrical and Electronics Engineering), pp. 250–259.
15. Mamrak, S. A., D. Leinbaugh, and T. Berk, "Software Support for Distributed Resource Sharing," *Computer Networks and ISDN Systems*, **9**(2), 91–107 (1985).

16. Brownbridge, D. R., L. F. Marshall, and B. Randell, "The Newcastle Connection, or UNIXes of the World Unite!," *Software—Practice and Experience*, **12**(12), 1147–1162 (1982).
17. Litman, A., "The DUNIX Distributed Operating System," *Operating Sys. Rev.*, **22**(1), 42–51 (1988).
18. Walker, B., G. Popek, E. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System," Proc. 9th ACM Symp. on Operating System Principles, Bretton Woods, N.H., October 10–13, 1983, *Operating Sys. Rev.*, **17**(5), 49–70 (1983).
19. Du Bois, P., and D. Mecozzi, "LINCS Guest File Server for the Cray-1," Lawrence Livermore National Laboratory, Livermore, Calif., working document, March 10, 1986.
20. Marques, J. A., J. P. Cunha, P. Guedes, N. Guimarães, and A. Cunha, "The Distributed Operating System of the SMD Project," *Software—Practice and Experience*, **18**(9), 859–877 (1988).

The Automounter

Brent Callaghan

Tom Lyon

Sun Microsystems, Inc.
2550 Garcia Avenue.
Mountain View, Ca. 94043

ABSTRACT

This paper describes the automounter – an automatic filesystem mounting service distributed with Sun Microsystems version of the Unix® operating system (SunOs). The automounter detects access to remote filesystems and mounts them on demand. This action is transparent to users and programs. Automounted filesystems are automatically unmounted after a period of inactivity. The map files that control the automounter can specify multiple locations for filesystems replicated across a network and can describe mount hierarchies. Automount maps can be administered on a single machine through local files or across a Yellow Pages domain.

1. Introduction

The automounter was originally developed as a component of Sun's Network Software Environment (NSE™) [1]. An important requirement was for seamless access to files whether they be on a local disk or on a remote server. A file anywhere on the network could be accessed with a Unix pathname, leaving the automounter the task of locating the file and mounting the filesystem containing it. The utility of the automounter was not confined to the NSE – it was quickly found to be a useful service in a general Unix environment, particularly in a large network where it was neither practical nor desirable to mount every exported filesystem from every server. The automounter is now a service available in SunOs version 4.0. As a user-level server for Sun Microsystem's Network File System (NFS™) [2], the automounter does not require any explicit support from the Unix kernel. It can be compiled and run on any version of Unix that supports the NFS protocol.

2. The Automount Server

The automount server is a daemon that provides NFS service at one or more mount points in the filesystem. The Unix kernel uses remote procedure calls to communicate with the daemon, just as it would for a remote NFS server. At any of its mount points, the daemon can intercept a request to access a remote filesystem, mount it if it's not already mounted, and return a symbolic link to the mount point.

As a simple example, consider */src* as an automounter mount point that provides access to source trees. Upon a reference to */src/bsd* the kernel would send an NFS *lookup* request to the automounter for the pathname component *bsd*. The automounter would lookup *src* in a source tree map, find the location of the corresponding source tree *server:/directory*, create a mount point in its */tmp_mnt* directory, and mount the source tree. To the original *lookup* request in */src*, the response would be a symbolic link to the mount point in */tmp_mnt*. The program that made the reference to */src/bsd* would have no way of knowing that the symbolic link did not exist prior to the *lookup* and that the source tree was not previously mounted.

2.1. Automount Behavior

The automounter needs to support only a small subset of the NFS protocol. The required subset depends on whether the automounter is emulating a symbolic link or a directory of symbolic links (Figure 1 overleaf). In particular, the automounter does not need to support *read* or *write* requests. It exists only to provide a name binding and mounting service. An Automounted file system is mounted within the */tmp_mnt* directory. The automounter merely creates a symbolic link to a mount point within */tmp_mnt* and hands it back to the kernel in response to a *readlink* request. From then on the kernel accesses the file system through the actual server.

At startup the automounter opens a UDP socket and registers it with the *portmapper* service as an NFS server port. It then forks off a server daemon that listens for NFS requests on the socket. The parent process proceeds to mount the daemon at its mount points within the filesystem. Through the *mount* system call it passes the server daemon's socket address and an NFS *filehandle* that is unique to each mount point. The daemon uses the filehandle to identify the mount point that is the source of subsequent requests from the client (kernel). Once the parent program has completed its mounts, it exits. The server daemon serves its mount points using one of two emulations at each mount point: symbolic link and directory of symbolic links.

2.2. Emulations

Symbolic Link

This emulation uses an entry in a *direct map*. In a direct map each entry has a pathname for an automount mount point, a remote filesystem location and mount options that correspond to this mount point. The automounter responds as if there is a symbolic link at its mount point. In response to a *getattr* request the automounter describes itself as a *symbolic link*. When the kernel follows with a *readlink* the automounter returns a path to the *real* mount point for the remote filesystem in */tmp_mnt*.

Directory of Symbolic Links

This emulation uses an *indirect* map. In response to a *getattr* request, the automounter describes itself as a *directory*. When given a *lookup* request it takes the name to be looked up and searches the indirect map. If it finds the name in the map, it returns the attributes of a symbolic link. In response to a *readlink* it returns a path to the mount point in */tmp_mnt* for this directory entry. A *readdir* of the automounter's mount point returns a list of entries that are currently mounted.

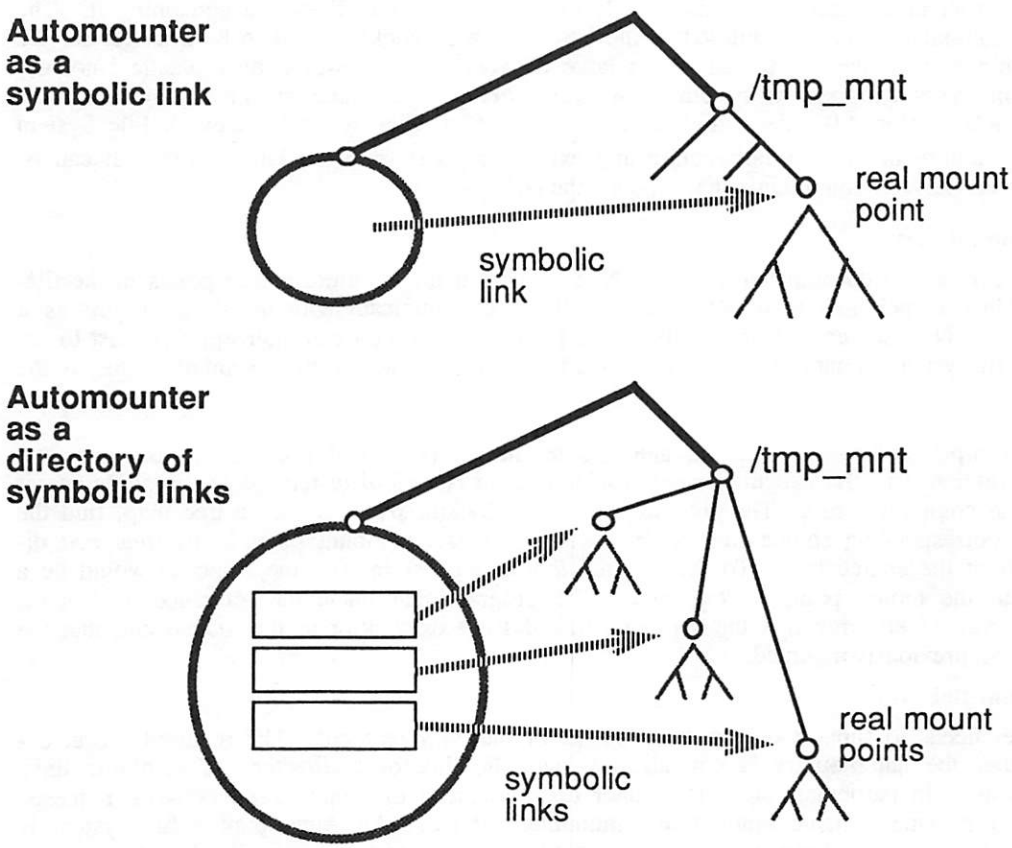


Figure 1

As an NFS server, the automounter sees only one component of a path to a remote filesystem through its mount point. Since it has no way of knowing in advance which exported filesystem will be accessed on the server, it must mount *all* the server's exported file systems. This may dismay casual users who notice that a reference to single filesystem through the *–hosts* map also mounts many unwanted filesystems. In practice this is not too much of a problem. A dozen mounts can be done in just a few seconds. The *–hosts* map is particularly useful for casual browsing of servers around a network. A specialized map should be created if frequent access to a specific file system is required.

The *–passwd* map was included to provide easy access to user's home directories around the network. It works only for a specific format of home directory path that includes the hostname of the server that exports the home directory. At Sun we use the format "/home/*server*/*loginname*". Given the login name of a user, the automounter makes a call to *getpwnam()* to get the user's home directory path. It uses the server name in the middle component of the path to build an internal map entry:

```
loginname          -ro,nosuid  server:/home/server:loginname
```

As an additional feature, given a *lookup* of the tilde "~" character, the user's *uid* is extracted from the credentials passed with the NFS request and the home directory path is obtained with a call to *getpwuid()*. A symbolic link containing a reference to a mount point for the *–passwd* map and a tilde as the next component would point back into the user's own home directory. This could be used as part of a scheme to move mail files from */usr/spool/mail* into users' home directories.

4. Administration

The automounter, its mount points, and maps can be administered on a single host through the use of regular files or across a whole Yellow Pages domain with YP maps. Every client that hosts an automounter must have an entry in its */etc/rc.local* file to start the automounter at boot time.

4.1. Local Administration

We have used the convention of putting maps that are files in */etc* and use a prefix of "auto." on the map name. In the absence of the Yellow Pages the *–hosts* and *–passwd* built-in maps would continue to work but only for entries in the */etc/hosts* and */etc/passwd* files.

A file map entry can be a reference to a YP map. If in the course of scanning a local map for key, the automounter finds a key with a prepended plus "+" sign, it treats the key as the name of a YP map to be consulted. This feature may be used to interpose map entries that are specific to the host before the YP map is consulted.

```
bsd4.2          berk:/usr/src:bsd4.2
bsd4.3          berk:/usr/src:bsd4.3
+auto.src
*               &:/usr/src
```

The map above provides access to various source trees. If the key is not `bsd4.2` or `bsd4.3` the YP map `auto.src` is consulted for the key. If it's not found there either it is caught by the "catchall" entry which assumes that the key is a server name and attempts to mount `/usr/src` from that server.

4.2. Yellow Pages Administration

Given a map name, the automounter first checks for a local file. If it's not a file it assumes that the name refers to a Yellow Pages map. The Yellow Pages service is not *required* by the automounter, but when it's available it can be used to administer the mount points for clients across an entire YP domain. Changes in the locations of file systems in a network need to be reflected only in the appropriate YP maps. The changes will be effected transparently on the clients who need not be aware of changes in mount location, mount options or the addition of new map entries.

A local map file can be converted to a Yellow Pages map by the YP administrator. A "catchall" entry will continue to work in a YP map. If a lookup for a key in a YP map fails, the automounter tries again using a "*" key and uses the catchall entry if one exists in the map.

4.3. Auto.master

At startup, the automounter checks for the presence of a YP map called *auto.master*. The syntax here is not that of the direct or indirect maps. Each entry contains a mount point, a map name, and optional default mount options for the map.

```
# Mount point    Map          Mount options
/-               auto.direct  -ro,intr
/home            auto.home    -rw,intr,secure
/net             -hosts
```

Changes made to this map by the YP administrator will affect every client in the YP domain that uses the automounter. Since *auto.master* is read only when the automounter is started up, any changes made to the map will not be effective until the automounter is restarted. A client is not forced to use the *auto.master* map either in whole or in part. Additional mount points and maps can be specified by an individual client either on the command line or in a file. A client can cancel a mount point in *auto.master* with a *–null* map on the command line. The entire auto.master can be disregarded by setting the *–m* flag on the command line.

A system administrator can exercise a great degree of control over the NFS mounts of each client. File systems can be added and moved about the network without the knowledge of the clients. Only the YP maps that control their automounters need to be updated.

5. Future Work

The current implementation requires a response to be sent in reply to an NFS request before the next request can be serviced. For most RPC services this doesn't present too much of a performance problem if requests can be serviced quickly. The automounter can respond quickly to NFS requests that reference cached symbolic links but it can be subject to substantial delays if a request requires a mount from a server that is slow or is not responding. Not only will the current request be delayed, but *all* requests will be delayed until the automounter responds. This unevenness of response time could become intolerable on a large multi-user machine with a single automounter daemon. The problem could be somewhat alleviated by forking off a separate daemon to serve each mount point at the cost of increasing memory usage. A solution we would like to pursue is to make the automount daemon support multiple threads of execution according to a Lightweight Process model[4]. This would allow the automounter to handle a number of NFS requests concurrently.

The locations in an automount map currently use hard-coded server and exported filesystem names. We would like to allow a filesystem location to be given as a name that could be mapped to a location (server:/dir) by a name binding service. This facility would allow individual servers in a network to "advertise" their exported filesystems.

The *-hosts* map permits easy access to all the *exported* filesystems from a server. This is commonly misunderstood to mean "access to the *mounted* filesystems of any host". This map cannot be used to browse the mounted filesystems of a diskless machine. We would like to extend the mount protocol to allow the automounter to find out what filesystems a host has mounted and allow it to reproduce those mounts. This facility would greatly improve the network transparency offered by the *-hosts* map.

The mount points served by the automounter are fixed at startup. The only way to add new mount points is to terminate the automount daemon with a *kill* command and start it up again. We would prefer to be able to offer uninterrupted automount service while such a change is made. An alternative structure for the automounter, would be to split it into two commands: a command that simply forks the server daemon with no mount points assigned, and a process invoked through the *mount* command that could mount the daemon at any given mount point and assign a map.

6. Acknowledgments

The development of the automounter has been spurred and encouraged by many people within Sun. Brad Taylor wrote the user-level NFS server skeleton on which the automounter was based. Bob Gilligan, Bill Shannon, Bob Lyon, Daniel Steinberg, Carl Smith, John Pope, David DiGiacomo, Marty Hess, and Dave Brownell all suffered early versions, described the bugs and provided many useful suggestions.

REFERENCES

[1] "Introduction to the NSE", Sun Microsystems, Inc. (1988)

[2] R. Sandberg *et al*, "Design and Implementation of the Sun Network Filesystem", *USENIX Conference Proceedings*, Portland, Summer, 1985.

[3] P. Weiss, "Yellow Pages Protocol Specification", Sun Microsystems, Inc. Technical Report, 1985.

[4] J. H. Kepecs, "Lightweight Processes for UNIX Implementation and Applications", *USENIX Conference Proceedings*, Portland, Summer, 1985

[5] David Hendricks, "The Translucent File Service", *EUUG Conference*, Portugal, 1988.

Improving the Performance and Correctness of an NFS Server

Chet Juszczak

Digital Equipment Corporation
110 Spit Brook Road ZK03-3/U14
Nashua, New Hampshire 03062
(603) 881-0386
chet@decvax.dec.com

ABSTRACT

The Network File System (NFS) utilizes a stateless protocol between clients and servers; the major advantage of this statelessness is that NFS crash recovery is very easy. An NFS client simply continues to send a request until it gets a response from the server. However, this client retry model also has disadvantages: a server can receive multiple copies of the same request. The processing of duplicate requests is an expense of server effort that is better spent elsewhere. Worse than that, it can result in incorrect results. This paper describes a *work avoidance* technique that utilizes a cache on the server to avoid the needless processing of duplicate client requests. An implementation of this technique has resulted in a significant increase in server bandwidth. A beneficial side effect is that it can help avoid the destructive re-application of non-idempotent operations. It can be used in any NFS server implementation, requires no client modifications, and in no way violates the NFS crash recovery design.

1. Introduction

The Network File System (NFS)[1] has become a standard in the UNIX[2] industry. The NFS utilizes a stateless protocol between clients and servers. This means that an NFS server is not required to keep any information (state) about a client request after it has been performed. Each NFS request contains all the information necessary for the server to perform an NFS operation [1].

The major advantage of this statelessness is that NFS crash recovery is very easy. Neither client nor server must detect the other's crashes. Since a server has no state information to maintain, there is nothing for it to throw away after a client crashes. Likewise, there is no state information to re-build when the server returns after a crash. An NFS client simply continues to send (retransmit) a request until it gets a response from the server. (Optionally, the client can give up after a number of retries specified at mount time.) This client retry model solves other service problems such as network disruptions, data loss at the server's network interface, and overflow of the server's input queue.

[1] NFS is a trademark of Sun Microsystems, Inc.

[2] UNIX is a registered trademark of AT&T.

However, the NFS client retry model also has disadvantages. To an NFS client, a server that is down simply looks like one that is slow to respond. Problems arise because a server that is slow to respond simply looks like one that is down.

Due to retransmitted requests, a slow or busy server can receive multiple copies of the same request. In fact, even the quickest of servers can receive duplicate requests from an impatient client. The processing of these duplicate requests is a waste of server effort that is better spent servicing other requests, perhaps from other clients. Worse than this inefficiency, duplicate request processing can result in incorrect results (affectionately called 'filesystem corruption' by those not in a filesystem development group). The *lost writes* seen at MIT by Project Athena [2] are the result of an NFS server processing duplicate client requests for a file truncation operation.

Included in the Ultrix[3] V2.2 diskless workstation project was the goal of improving the performance of our NFS implementation. Since all of the diskless workstation's files would be accessed via NFS, its filesystem performance would be equivalent to its NFS performance. To a large extent, a client's NFS performance is determined by its server's performance.

The write operation is the most costly of all NFS server operations (see *NFS Writes*, below). A single client can have multiple outstanding write requests, and a diskless client generates more NFS write requests, on average, than a diskful one. This follows from using NFS for paging, swapping, and */tmp*.

An investigation of diskless client performance [3] has shown that a heavy write load results in poorer client performance than loads of other types. Detailed server analysis indicated that when under heavy write load, servers expended considerable effort servicing duplicate requests.

A *work avoidance* technique is described that utilizes a cache on the server to avoid the needless processing of duplicate client requests. An implementation of this technique has resulted in a significant increase in server bandwidth, especially with low end server configurations. A beneficial side effect is that the destructive re-application of a duplicate operation is much less likely.

There is no need to re-build the previous contents of the cache after a server crash. Therefore, the use of this cache violates neither the statelessness of the NFS protocol nor its crash recovery design. No NFS client side modifications are necessary, although one change is suggested that makes the technique more effective (see *Results and Recommendations*).

2. Background

This section contains some necessary background information on various NFS topics. References to "typical" NFS clients and servers refer to implementations of NFS derived from the 4.3BSD based kernel implementation available from Sun Microsystems, Inc. Similarly, the term *reference port* will refer to this implementation. Our work was done with version 3.2 of the reference port. The following characterizations may apply to other NFS implementations as well.

2.1. NFS Clients and Servers

NFS client systems range in size from small workstations to large multi-processor timesharing machines with hundreds of users.

A typical NFS client system will retransmit a request if it has not received a response from the server for that request within an interval of time that defaults to .7 seconds. This interval is implementation dependent and can also be specified as a parameter when

[3] Ultrix is a trademark of Digital Equipment Corporation

the filesystem is mounted. The client will retransmit again (and again, ...) if no response is received. The time interval is increased using a backoff algorithm (next interval = current interval X 4) up to a ceiling value (60 seconds) after each successive timeout. This level of impatience that a client has for a server is determined solely by the client and is not dynamically adjusted based on past server performance.

The number of retries within a *timeout cycle* is implementation dependent and is also a mount time parameter. The reference port sets the default number of retries in a timeout cycle to three. With a *soft* mount (a mount option), if a response is not received for a request within the first timeout cycle, then the client operation (system call) fails. With a *hard* mount, the request is retransmitted until a response is received. With a hard mount, a single NFS request may span more than one timeout cycle before it receives a reply. The backoff algorithm described above continues to increase the timeout interval across timeout cycles.

The client RPC (*Remote Procedure Call* [4]) layer assigns a transaction ID (*xid*) to each outgoing request. Duplicate requests within the same timeout cycle will have the same *xid*. Duplicate requests will have different *xids* when the number of re-transmissions exceeds the number of re-tries in a timeout cycle. Version 2 of the NFS protocol does not define an NFS transaction ID that is unique to each NFS request from a given client. This makes it impossible for a server to reliably determine whether two requests are actually duplicates. [It is not enough to know that two requests appear to be the same (e.g. a write to the same location in the same file), since they could have been generated by two separate client processes in sequence.]

The client system can have multiple outstanding read and/or write requests. A client process blocks whenever a read or write request cannot be satisfied locally and must be processed by the server. When it blocks, another process can run; that process may also generate a read or write request. A single process can have multiple outstanding read and/or write requests if the client system is running NFS block I/O (*biod(8)*) daemons. These daemons perform client read-ahead and write-behind functions asynchronously, allowing the client process to continue execution. Each outstanding request will time out individually; each can result in a retransmission. Clients discard duplicate responses (the second response received for a single request) as unsolicited input, but they are counted as a *badxid*, and available via *nfsstat(8)*.

NFS server systems range in size from small machines with a single, slow disk to large multi-processor machines with disk farms.

A typical NFS server system simply waits for work to appear on an incoming request queue. This queue is the socket buffer allocated for the NFS socket. Incoming requests are converted into a form understandable by the local filesystem routines that actually perform the work of getting data to/from a disk. The incoming request queue is of fixed size. If the queue fills (requests coming in faster than they can be processed) then some incoming requests may be lost.

The amount of work that a server can perform is called server bandwidth. It is usually not limited by CPU speed, but by network interface and/or the disk subsystem performance. Server bandwidth is sometimes measured in a general manner, e.g. NFS operations/second, and sometimes specifically, e.g. read or write speed in Kbytes/second. A typical server does not prioritize incoming requests based on type of request or originating client.

Ignoring access rights and security, an NFS server has limited control over how it is used. An administrator decides:

- Whether to serve or not. A system serves by running *nfsd(8)* daemons.
- What to serve. A server only allows operations on *exported* filesystems.

- How many *nfsd* daemons to run. This controls the number of NFS requests that the server can work on concurrently. It also controls the amount of local resources (e.g. disk bandwidth) that is available for remote use (versus use by local processes).

The server depends upon its clients to attenuate their request loads as it becomes heavily loaded (i.e. the aggregate load is coming in faster than it can be processed). However, nothing makes a particular client implementation act kindly towards a server. There is no way to enforce the client backoff scheme described above. Most implementations allow a client administrator (or workstation user) to run as many block I/O daemons as they wish, and to mount with retransmission timeout values that are very small. When faced with one of the latest generation workstations armed with a suitable workload, the performance of even the most powerful server configurations can degrade drastically.

2.2. NFS Writes

Since an NFS server is bound by a stateless protocol, it must commit any modified data to stable storage before responding to the client that the request is complete [1]. If a server is not following this rule, then it is not living up to its part of the agreement implicit in the NFS crash recovery design. An asynchronous operation carries with it the promise to fully complete that operation at some later time. Without a way to recall past unkept promises, a server cannot make them. The protocol contains no provisions for recalling past promises (which is precisely why crash recovery is so easy). Therefore, a server must complete each data modifying operation fully (synchronously) before responding .

For each remote write request, at least one, and possibly two or three synchronous disk operations must be performed by the server before a response can be sent to the client indicating that the request has been completed. At the very least, the data block in question must be written. If the write increased the size of the file, or on-disk structures have changed (e.g. adding a direct block to fill a "hole" in the file), then the block containing the inode must be written. Finally, if an indirect block was modified, then it too must be written before responding.

The reference port makes a special case for the file modify time in the inode. If modify time is the only item changed in the inode as a result of a write operation, i.e. a write to a previously allocated block, then the inode update to disk is performed asynchronously. This is one promise that the server may not keep; the risk is taken for the benefits of better performance.

2.3. Duplicate Requests

Duplicate requests (sometimes called delayed retransmissions) are part of the NFS crash recovery design. A server can receive a duplicate request while performing the original request. Multiple copies of a request can be received and placed on the input queue before any are processed. If a client is preparing to retransmit when the response it wanted is received, the server will still be sent a duplicate request.

2.4. Non-Idempotent Operations

When used in a database context, the term *idempotent* is used to describe transactions that can be applied more than once without any ill effects. Inquiry transactions are *idempotent.* Debit and credit transactions are *non-idempotent.*

When used in an NFS context, the term can be used to distinguish between request types. An *idempotent* request (e.g. read) is one that a server can perform more than once without side effect. The side effects caused by performing a duplicate request can be classified as destructive and non-destructive.

To simplify the following discussion, we will only consider the case where the file in question is being accessed by a single remote client and not being shared between multiple remote clients and/or processes local to the server. Of the sixteen request types in version 2 of the NFS protocol (the only version in production use today), nine are *non-idempotent.* These are:

Table 1. Non-Idempotent NFS Operations

Operation Name	Description
create	create a file
remove	remove a file
link	create a link to a file
symlink	create a symbolic link
mkdir	make a directory
rmdir	remove a directory
rename	rename a file
setattr	set file attributes
write	write to a file

The first seven operations in Table 1 are obviously *non-idempotent.* They cannot be reprocessed without special attention simply because they may fail if tried a second time. The create request, for example, can be used to create a file for which the owner does not have write permission. A duplicate of this request cannot succeed if the original succeeded. Similarly, you can only successfully remove a file once; if permission was granted the first time, a second try cannot succeed. This type of scenario is not destructive, but is a nuisance for the server implementation to sort out.

Another scenario, one that does have destructive side effects, involves retransmitted *non-idempotent* requests and a race condition between *nfsd* daemons on the server. In Example 1, the client runs a process that creates a file and writes one block into it. The server is running two *nfsd* daemons. Client and server activities are shown at a number of points on a time line; the points are not equally spaced.

Example 1. Destructive Non-Idempotent Scenario

Time	Client Activity	Server Activity
t0	process starts	idle
t1	transmit create request c(0)	idle
t2	wait for create response	receive c(0), schedule *nfsd1*
t3	retransmit create request c(1)	*nfsd1*: complete c(0), truncate file, send create response
t4	receive create response process resumes	receive c(1), schedule *nfsd1*
t5	transmit write request, w(0)	*nfsd1*: starts but blocks on a system resource
t6	wait for write response	receives w(0), schedules *nfsd2*
t7	wait for write response	*nfsd2*: complete w(0) send write response
t8	receive write response process completes	*nfsd1*: complete c(0), truncate file, send create response
t9	receive create response and discard it	idle

In Example 1, the server processes two create requests and one write request. The net effect is a zero length file; the write has been lost. This problem has been seen at MIT [2].

There is a variant of the scenario in Example 1 that involves no *nfsd* block on a system resource (t5). If both *nfsd1* and *nfsd2* are scheduled, one with the write and the other

with the duplicate create, then scheduler vagaries can make their order of execution unpredictable.

The setattr and write operations are not as obvious as the first seven listed in Table 1. They are destructive only if re-applied after some other intervening operation. Setattr can be used to truncate a file; it can be used instead of create in a scenario similar to Example 1. Write would appear to be *idempotent*, but there is a curious destructive case here as well. If a duplicate write request is applied in a server *nfsd* race after a file truncation, then the file size is non-zero and the file contents are binary zeroes + the block of write data.

2.5. Reference Server Transaction Cache

The 4.3BSD based kernel implementation of NFS that is available from Sun Microsystems, Inc. features a cache of recently processed transaction IDs (*xids*) on the server. This cache is implemented in the kernel RPC layer, not in the NFS layer.

The server NFS layer uses this cache to store the *xids* of recent requests that have succeeded. The scope of the cache is five types of transactions: create, remove, link, mkdir, and rmdir. The cache is accessed by hashing the *xid* into an array of lists. The cache entries are re-used in a round-robin fashion. If an operation fails, the cache is used to help determine if the failure was due to the request being a duplicate of one that previously succeeded. If it is, then a positive response is returned to the client. Used in this manner, the cache helps to sort out the non-destructive type of behavior described above, but not the destructive behavior.

Destructive behavior is not prevented for two reasons:

- The cache is not used or consulted for the setattr or write operations.
- For the operations where the cache is used, it is searched only after an operation has already been performed, and only then if it has failed.

3. Wasted Server Effort

As mentioned earlier, we chose to look at servers under write load for areas where server performance could be improved. An NFS server implementation (for UNIX) is an order of magnitude simpler than the NFS client implementation. That made it a simpler place to look for a quick improvement. Also, a server can be used by many different client implementations; improving our client implementation didn't necessarily improve conditions for our servers.

NFS servers were placed under load and their activities were logged. The logging was done by modifying the server kernel to write information of interest to the existing Ultrix error log facility. An existing kernel interface for writing to this error log was used. The *nfsstat(8)* program was used on the client to measure the total number of write requests and the number of duplicate responses (*badxids*).

An examination of the server log showed that when the server's response slowed, the client would "ask again", as designed. This duplicate request, or retransmission, was added to the already heavy load on the server. Performing the operation a second time helped guarantee that it would be slow in responding to some other request, which might then result in a duplicate of that one, and so on.

Our *reference port* based NFS server implementation didn't distinguish the second request for work that had already been done from the first request and performed the request a second (or third!) time.

When the workload is "expensive" to perform (writes are the most expensive [3]), it is relatively easy to create this wasteful server scenario. One μVAX-II[4] client system running 4 *biods* can have up to 5 concurrent write requests from a single user process. A

[4] VAX and μVAX are trademarks of Digital Equipment Corporation.

simple test program was used that wrote 1 Mbyte to a file via 128 8 Kbyte write requests. The file was always 1 Mbyte in size when the program started; the program emptied the file when it was opened and proceeded to extend it to the 1 Mbyte final size. A write that extends a file is the most costly of write requests (see *NFS Writes* above).

In the tables that follow,

- 'small server' is a dedicated μVAX-II with an RD53 (70 Mbyte Micropolis 1325D Winchester) running 4 *nfsds*.
- 'large server' is a dedicated VAX 8550 with RA81 450 Mbyte disk running 4 *nfsds*.
- 'write requests' is the number of 8 Kbyte writes generated by the client.
- 'duplicate writes' is the number of retransmitted client write requests.
- 'excess writes transmitted' is a measure of the excess data sent from the client to the server.
- 'duplicate writes processed' is the number of duplicate writes performed by the server.
- 'excess writes processed' is a measure of wasted server bandwidth.
- 'write speed' is a client measure (in Kbytes/second) of throughput; it is 1 Mbyte divided by the time needed to open, truncate, write, and close the file.

When the test program is run, a minimum of 128 write requests must be made; if more are generated, then they are duplicates. Likewise, 128 write requests must be processed by the server; processing more is a waste of server bandwidth. The initial timeout interval used was .7 seconds. Twenty iterations of this test program were run and statistics gathered. Table 2 contains the averaged results.

Table 2. Write Load Test Results

	small server	large server
write requests	190	141
duplicate writes	62	13
excess writes transmitted	48%	10%
duplicate writes processed	61	13
excess writes processed	48%	10%
write speed (Kbytes/sec.)	35	75

These results showed that significant server bandwidth was spent performing duplicate writes. This relatively simple test immediately indicated an area upon which to focus. If the server could be made to spend less bandwidth on "useless" work when under write load, then it would have more bandwidth for "useful" work.

4. Server Modifications

The *reference port* RPC transaction cache described above was used as a starting point.

The scope of the cache was increased from: create, remove, link, mkdir, and rmdir to include all request types. More information was added to each cache entry: an in-progress flag, a transaction completion code, and a time stamp. The size of the cache was kept at the reference port size of 400 entries. The entries are re-used in the same round-robin manner as in the reference port. That is, entries age equally and can be referenced right up to the time of re-use.

The major change is in how the NFS layer uses the cache. The reference port performs an operation and checks the cache afterwards only if it fails and it is one of the types within the scope of the cache. This approach was changed in the following ways.

4.1. Requests In-Progress

A check of the cache on every incoming request was added as one of the very first NFS layer operations. If the transaction is in the cache, and marked in-progress, then the request is counted and quickly discarded without any response. If the transaction is not in the cache, it is added and marked in-progress.

The overhead associated with a fast cache check pales in comparison to the overhead associated with performing even the fastest request and transmitting a (useless) response. When under load, the check is saving much more than it costs. When not under load, the cost is negligible. As for throwing the request away: it is, after all, a duplicate and already getting attention; a response will be forthcoming.

4.2. Requests Recently Completed

If a duplicate request is received within a certain time interval (called *throwaway window*) and the original request was processed and successfully completed, then the request is counted as a duplicate and quickly discarded without a response. A duplicate of a request that earlier failed is re-tried. Experiments with various intervals showed that a *throwaway window* of from 3 to 6 seconds was sufficient to drop the processing of duplicate requests to nearly zero.

If a client asks a server to repeat an operation, then either the response was lost, or more likely, the duplicate request passed the response in transit, i.e. the client has actually gotten the original response by the time the server starts to process the duplicate. In the former case, the client will simply keep asking; after the *throwaway* interval has expired, the server will perform the request. The latter case is exactly what we are looking for; we truly want to throw this request away. It isn't clear whether it is better to retry requests that have previously failed or not. Failures don't happen often and are usually processed quickly. By re-processing, the old server behavior was kept for failure cases. Therefore, nothing was broken that was not previously broken. [There is a potential, but unlikely, problem here: the client may get a failure response from the original request and the server replays a duplicate which for some reason now succeeds. The client thinks the operation failed, when in fact it has succeeded. This can cause client/server cache consistency problems.]

4.3. Non-Idempotent Operations

Special treatment is given to the *non-idempotent* operations (including setattr and write). History on these operations is kept, namely completion status and a time stamp when the operation completed. When one of these operations is completed, its cache entry is updated with completion status and the appropriate inode time (either "modified" or "changed"). When a duplicate is received for a request that earlier succeeded, and the inode indicates that the file has not changed since that time by comparing its time with the time in the cache entry, then a successful response is returned to the client. This will occur until the cache entry is re-used and the cached information is lost. After that time, a duplicate request will be processed as a new one.

The goal was to avoid re-processing these duplicates as long as possible. Writes are very expensive and the *non-idempotent* operations can be dangerous. If the earlier request has succeeded and the file has not since changed, then it seems clear that to return a response is correct. What to do for earlier failures is less clear, but (like above) by re-processing them, old server behavior was maintained.

5. Results and Experiences

When the write tests described earlier were run using a modified server, the number of duplicate writes performed by the server dropped to zero (the server was using a *throwaway window* of six seconds). Since the server was doing less wasteful work, it responded faster to the necessary work and the write throughput as seen by the client increased. Since the amount of bandwidth wasted was greatest on the small server, that is where the improvement was greatest. Table 3 compares the earlier results (from Table 2) with the averaged results obtained using a modified server.

Table 3. A Comparison of Write Load Test Results

	small server	modified small server	change	large server	modified large server	change
write requests	190	190		141	141	
duplicate writes	62	62		13	13	
excess writes transmitted	48%	48%		10%	10%	
duplicate writes processed	61	0	-100%	13	0	-100%
excess writes processed	48%	0%	-100%	10%	0%	-100%
write speed (Kbytes/sec.)	35	48	+37%	75	83	+11%

The real gains are not necessarily seen from a single client, but a group of clients that take advantage of the server bandwidth increase and better response to load conditions.

A very beneficial side effect is that there have been no reports of the destructive behavior caused by the re-processing of *non-idempotent* operations. These operations, as a whole, are expensive to perform, but their frequency (except for write) is relatively low. The major reason for handling them specially is their destructive potential.

The server modifications described here were integrated into Ultrix V2.2, which has been in production use since early 1988.

In Ultrix V3.0 the client RPC layer was modified so that the caller (NFS) could define the transaction ID (*xid*). In this way, the NFS layer can control the *xid* and make sure that it is unique for each request, even across timeout cycles. This in turn helps make the server transaction cache even more effective. More work needs to be done comparing the effectiveness of different cache sizes over varying workloads.

6. Conclusions

NFS servers were placed under heavy write load and the results (Table 2) showed that the system was unstable in response to this load. A heavy write request load resulted in retransmitted write requests that resulted in a heavier load. The server modifications using the *work avoidance* technique described earlier reduced the positive feedback effects of duplicate client requests. These results are displayed in Table 3. The modified server responds to heavy load by simply trying to throw some requests away. This notion didn't set well at first. But the problem under observation was a slow server looking like a downed one to an impatient client; it seemed ironic (and amusing) that it might help if the slow server started selectively acting like a downed one and discarding requests. Throwing a duplicate request away opens up server bandwidth for other work; not responding reduces the amount of network traffic. The NFS client retry model will take care of server responses that are lost on the network or by the client. NFS is usually used on LANs, where network losses are infrequent. Therefore, delays due to this scheme should be rare.

An implementation of this technique resulted in a significant increase in server bandwidth, especially with low end server configurations. A very beneficial side effect is that the destructive re-application of duplicate *non-idempotent* operations is made much less likely. There is no need to re-build the previous contents of the cache after a server

crash. Therefore, the use of this cache violates neither the statelessness of the NFS protocol nor its crash recovery design.

The load tests showed that the server's input queue worked very well (at least with this selective type of load). When under heavy write load, very few requests were lost (seen by comparing the number of duplicate requests generated by the client with those processed by the server). The limiting factor was not an input queue that was too small. An analogy can be drawn here between NFS server congestion and network congestion. John Nagle [5] has shown that long (even infinite) queues won't help a network gateway with an overload; the load must be shed in some way. The modified server screens its load and sheds requests that it is currently processing or has recently completed.

The mechanisms leading to congestion in datagram packet-switching networks are similar to those that lead to congestion in stateless (non flow-controlled) RPC based file servers. Van Jacobson [6] has shown that an effective answer to network congestion is for the client to detect it and back off on its request rate. Raj Jain [7] stresses the need to operate a system below the point of unstable positive feedback. An area that begs for future NFS work is the client side of the feedback loop. The NFS *reference port* client implementation does back off when retransmitting a given request more than once, but it does not use past server response characteristics to determine future initial timeout intervals.

7. Recommendations

This technique is recommended for use by all NFS server implementations. An NFS server selects neither the implementation of its client, nor its timeout characteristics. Even if NFS client timeout behavior is modified in some future implementations, current ones (using version 2 of the NFS protocol) will be used for years to come; performance gains are obtainable with them by using this technique. Likewise, the correctness improvements seen by this technique are useful for current client implementations. Version 3 of the protocol won't be defined until at least 1989; it may fix some of the *non-idempotent* operation problems, but won't be widely used until years after that.

NFS servers that use this technique can offer better performance and correctness to their clients. It is recommended that client implementations change their RPC layer to assign unique transaction IDs (*xids*) to NFS requests. This improves the effectiveness of the technique. It is further recommended that Version 3 of the NFS protocol include a transaction ID in each request.

8. Acknowledgements

The diskless performance work done by Charlie Briggs [3] was the takeoff point for this work. I further relied on Charlie's judgment, patience, and encouragement while writing this paper. John Dustin, Fred Glover, Joe Martin, Bob Rodriguez, Ursula Sinkewicz, and Mary Walker were draft reviewers; they offered many comments that greatly improved this paper's readability. Jeff Mogul made the insightful analogy between stateless file server congestion and datagram packet-switching network congestion [5][6][7].

9. References

[1] Russel Sandberg, et. al., "Design and Implementation of the Sun Network Filesystem", *USENIX Summer Conference Proceedings,* pp. 119-130, June 1985. Where it all started.

[2] Described by Dan Geer, MIT Project Athena, Nationwide File System Workshop, Carnegie Mellon University, Pittsburgh, PA, August 23-24, 1988.

[3] Charles Briggs, "NFS Diskless Workstation Performance", Digital Equipment Corporation Technical Report, February 24, 1988.

[4] Bob Lyon, "Sun Remote Procedure Call Specification", Sun Microsystems, Inc. Technical Report, 1984

[5] John Nagle, "On Packet Switches with Infinite Storage", in *IEEE Topics in Communications,* pp. 435-438, April 1987.

[6] Van Jacobson, "Congestion Avoidance and Control", in *Proceedings of SIGCOMM88,* pp. 314-329, Stanford, CA, August 1988.

[7] K. K. Ramakrishnan and Raj Jain, "A Binary Feedback Scheme for Congestion Avoidance in Computer Networks with a Connectionless Network Layer", in *Proceedings of SIGCOMM88,* pp. 303-313, Stanford, CA, August 1988.

MSS-II and RASH
A Mainframe UNIX Based Mass Storage System with a Rapid Access Storage Hierarchy File Management System

Robert L. Henderson
Alan Poston
GE - Western Systems
NASA Ames Research Center
Moffett Field, CA 94035
hender@prandtl.nas.nasa.gov

ABSTRACT

UNIX® on a large mainframe can provide hundreds of gigabytes of disk space. However, in a computer center with several super-computers, even that is not sufficient to support the user community. To support the users of the Numerical Aerodynamic Simulation facility, a national resource center, work is under way to develop a UNIX based Mass Storage System which will support files on disk and multiple levels of removable media.

Included in the Mass Storage System are a number of features to improve performance of the file and networking systems and new features for file hierarchy and volume management. This paper will address the storage hierarchy manager and the striping file system.

The high performance file system is a striped reliable file system that is optimized for large file sequential access without penalizing random access.

Unlike prior archival or migration systems which block the completion of the open system call until all of the data has been restored to disk; the rapid access archive and restore system provides for fast access to the data. The open call is not blocked. Instead, read or write access is blocked only when the data being accessed has not yet been restored to disk.

1. Introduction

This paper describes portions of the work to develop a second generation Mass Storage System now under way at the Numerical Aerodynamic Simulation facility located at NASA's Ames Research Center.

Included in the Mass Storage development are a number of projects to improve the capability and performance of the UNIX operation system and to provide the features necessary to use UNIX as the base for a large mass storage system. These projects include:

- A file archive and restore system (storage hierarchy manager) which provides rapid access to the data.
- A high performance striped file system which includes device failure recovery features.
- A file system capable of holding large files, greater than 2 gigabytes.

- A *safe* file system, that decreases the impact of a system crash or user error on files within the system and that includes a "trash can" feature.
- A removable volume management system, which extends the UNIX file system protection feature to tapes and other removable volumes.
- A data migration system which moves files from other systems to the mass storage system.
- Development of a network interface with very high data transfer rate.

This paper will address the storage hierarchy manager in great detail and the striping file system in some detail. Information on the other projects will be the subject of future papers.

NASA's Numerical Aerodynamic Simulation facility *NAS*, is a national supercomputer center dedicated to providing computational capability in fluid dynamics and related disciplines and to being a path finder in advanced, large-scale computer system capability.

A simplified picture of the NAS environment is shown in figure 1.

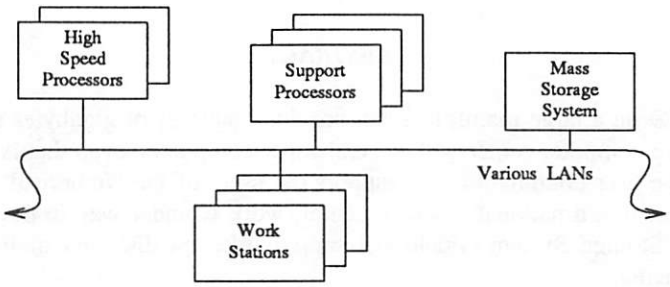


Figure 1 - NAS System

The current NAS environment[NAS88] consist of:

- Two Cray-2 supercomputers with 70 gigabytes of disk storage.
- A Cray YMP supercomputer with 53 gigabytes of disk storage.
- Convex and Alliant mini-supercomputers.
- A Connection Machine (32K processes) with a Data Vault.
- An Amdahl 5880 serving the dual roles of general user support processor and mass storage subsystem with 240 gigabytes of disk.
- 4 Vax 11/780 providing additional support and network processing.
- 35+ Silicon Graphic 2500 Turbo and 3030 IRIS graphic workstations.
- 40+ Sun and other workstations.
- Ethernet, NSC Hyperchannel, and Proteon networks.

The NAS environment is unique among supercomputer centers in that UNIX is the only operating system seen by the user. UNIX is run on all systems from the workstations to the supercomputers and on the mass storage system. All the systems are interconnected via one or more of several networks of varying performance. All networks use TCP/IP. The mass storage system appears as just another UNIX node on the network; in fact, the mass storage system is more of a very large network file server than a traditional mass storage system.

The NAS system serves in excess of 1000 users working on approximately 200 projects. The main scientific users use high capability graphics workstations to pre and post process data which is input or output to jobs run on the high speed processors (supercomputers). The typical supercomputer run may take up to 10 hours and produce from 100 megabytes to 2 gigabytes of data. It is expected that within a

year, the maximum file size will grow to 4 or more gigabytes.

2. The MSS-II System

The follow section presents an overview of the goals and requirements which influenced the design choices made in the MSS-II project.

2.1. Configuration

The MSS-II is based on the following configuration: Amdahl 5880 with 64 channels and 64 MB of main memory; 240 GB of disk, 8 IBM 3480 tape cartridges drives; two Storagetek 4400 Automatic Cartridge Library with 16 drives and 11,000 tape capacity; and network connections via NSC Hyperchannel and ethernet. UTS/580® version 1.2.2, Amdahl's port of System V.2, is the operating system.

2.2. Goals of the MSS-II System

The goals of the MSS-II project are to provide a network file storage system which:

- Appears to the user as a UNIX system.
- Handles file sizes up to 10 gigabytes.
- Provides total disk storage of at least 240 gigabytes, up to 600.
- Provides a minimum of 3 levels of files storage (on-line disk, tape in a robotics handler, and tape on the shelf); the middle level should provide 1.5 terabytes of storage.
- Provide a peak aggregate file to network transfer rate of 85 Mb/s (megabit per second; 10+ MB/s).
- Provide access to the first byte of data, from level 1 (disk) in 0.5 seconds and from level 2 (tape in robot) in 20 seconds.
- Ensure that no **single** software, hardware, or media **error** will result in loss of user's data.

It is apparent that the I/O performance requirements cannot be met without major changes to the traditional UNIX file system. The goal of multiple storage levels require an archive/restore system and robot support. Additionally, the time to access the first byte of data from a file stored in level 2 rules out the method used in most archive/restore systems for restoring files.

The safety issue, i.e. "... no single...error...", also introduces three interesting concepts. Because of the decreased reliability which comes with a large number of disk drives, the file system must include reliability code beyond *fsck(1)*. When files are archived to removable media, they must be copied to (at least) two volumes. To help protect the user from himself, *unlinked* files are moved to a *trash can* directory rather than being deleted. The file system reliability issue and the archive dual copy will be covered in more detail later in this paper.

2.3. MSS-II vs. Other Mass Storage Systems

One question which might arise is why another MSS, why not Common File System (CFS) from Los Alamos or the National Center for Atmospheric Research (NCAR) system.

CFS is perhaps the oldest of the existing mass storage systems having been in use since 1979.[CFS88] CFS serves as a centralized file server for a large heterogeneous collection of computers. The central server is based on MVS running on mid sized IBM type mainframe. It is connected to the other systems via Hyperchannel and uses either a special protocol or NSC's Netex. All file transfers pass through the mainframe. The central server implements its own file system with emphasis on security. A user interface is implemented on each of the connected systems. Users are restricted to doing "get" and "put" operations on their files.

The NCAR system is similar to CFS as it too runs on an IBM type mainframe under MVS, uses Hyperchannel, implements its own file system, and has a user interface on each connected system. The difference is that data does not flow through the mainframe. Commands to allocate space on disk or to position to the file on disk or tape go to the mainframe. However, data transfer is directly from the network to the device. To accomplish this direct connection, a special network adapter is required and each

connected computer in the user interface program must build and send an IBM-style channel program (I/O command sequence) to the various disk and tape devices. As with CFS, users of the NCAR system are restricted to "put" and "get" operations on their files.

While the above systems are very functional, there were several reasons why NAS chose to take a different approach. First, the MSS must have an UNIX interface. This might have been done by adding functionality on top of a system like MVS, but with great difficulty. Second, the MSS must be an integral part of the NAS network, using TCP/IP on whatever physical link existed now or in the future. Third, the design must not restrict the users of the MSS to "get" and "put" operations. The design of MSS-II does not restrict the user from performing any operation, such as *grep* on the MSS. This may save network traffic. NAS will leave the decision of command availability to the system administrator.

3. The Rapid Access Storage Hierarchy System

Since users of the NAS system keep their files for an extended period of time (forever) and since even in the MSS on-line space is limited, there must be a method of transparently moving files from disks and restoring them when they are needed.

The primary purpose of the Rapid Access Storage Hierarchy file manager, *RASH*, is transparently and optimally to manage the use of the various types of storage media in the Mass Storage System. This includes the movement of files between levels of storage; the archiving of files from disk to higher levels, the restoration or staging of files from higher levels (non disk) to the lowest level (disk), and the movement of volumes between the higher levels of the hierarchy. RASH will act to maintain a defined amount of free space on the file systems. RASH will also manage volume space within the higher levels of the hierarchy, ensuring space to migrate volumes up or down.

A second major goal of RASH is to provide the user with the fastest possible access to a file which has been archived. This "rapid access" takes the form of allowing access to any byte of data as soon as it has been restored to disk. This is the major point of departure between RASH and other archival or migration systems. This will be discussed in detail shortly.

The RASH project has other secondary goals:

1. RASH must do its part to live up the MSS-II promise that no single media error will result in the loss of data. Therefore whenever RASH archives a file, two copies are made to different volumes.
2. When archiving files, RASH groups files belonging to a single user together on volume sets which are separate from sets containing files of other users. This aids in the delivery of a user's files to the user should he wish physical possession of the file.

   Additionally, "small" files will be collected together on a set of volumes separate from "large" files. If a file will not fit on the current "small file set," that set is closed and a new set started. No attempt is made to do a "best fit" of the file onto older sets. This aids in temporal locality of files because files archived at the same time are on the same set. If they are accessed together later, the number of volumes accessed is minimized.

   The definition of large and small is left to the systems administrator, but is usually a function of the removable volume's capacity.
3. Modifications to the UNIX kernel must be made as cleanly as possible. This is tempered by one overriding factor: performance. Rapid access requires changes to the kernel beyond that required by other migration systems; specifically, the disk inode size must be increased to accommodate extra control fields.
4. In order to facilitate future changes to the database and to provide for ease in producing reports, RASH is interfaced to a commercial relational data base product.
5. Since the vast majority of space in a file system is occupied by file space, RASH limits itself to archiving regular files. No attempt is made to archive directories.

6. Files must be maintained on removable volumes in a standard interchange format. For tapes, the format is ANSI labeled with 40K data blocks.

3.1. RASH User Interface

RASH is intended to be as transparent as possible. The user need take no action to have a file archived (moved from disk to off-line storage) or be restored on-line. An archive subsystem under control of the system administrator selects which files are archived. An standard *open(2)* system call will result in the restoration of an archived file.

In fact, the only methods through which a user can determine if a file has been archived are by new features incorporated into the *ls(1)* command or a new system call, *rashcntl(2)*, which obtains additional status over and above what *stat(2)* returns. The ls command uses the rashcntl call.
When used with any of the "long" options, ls will indicate that a file is archived by showing a "m" character as the first character of the mode:

mrw-rw-rw- 1 hender npo 42135 Nov 10 14:49 outline

A new option, "-h", has also been added to the ls command. It will display the highest level of storage in which the file resides as a single digit just before the file type character:

2mrw-rw-rw- 1 hender npo 42135 Nov 10 14:49 outline

3.2. RASH Compared to Other Archive Systems

So far, the goals of RASH appear very similar to any archive/migration system, several of which are being developed for UNIX. The most notable of these is the BUMP project[BUMP88] at BRL and the US Naval Academy. Cray Research is also implementing a data migration system very similar to BUMP.

BUMP does transparent archive and restore operations. Additionally, BUMP does not require any inode size change. File migration is accomplished through a process called "pre-migration" where the content of the file's inode is copied into a second inode. The original inode file type is changed from *IFREG* to *IFMIG* and the disk address array is overloaded with a file handle. This file handle (based on the inumber) is used as the file name of the pre-migrated inode and as an key to the database. When the file contents has been archived onto secondary storage, the database is updated with the volume identifier, and the pre-migrated inode is released. Upon restoration (or staging) of a file, the pre-migrated inode is re-created and used as the basis of the copy operation. When the file has been completely copied back to disk, the pre-migrated inode's content is copied into the original inode, which is reset to type IFREG.

BUMP, like all other archive systems to date, blocks the open operation if the file is archived. The open operation remains blocked until the entire file has been restored to disk. This is the major departure of RASH from BUMP and all other archive systems.

3.2.1. RASH Rapid Access

RASH initiates the file restore as part of the open process. However RASH **does not block the open**. RASH maintains a new field in the in-core inode during the restore process; this field, the "current resident size" is the amount of data that has been restored to disk. It is updated by RASH as each block of data is copied from the higher level of storage to disk. As the user process accesses data, via *read(2)* or *write(2)* calls, that operation is blocked if and only if the data being accessed, *offset plus count*, is larger than the current resident size. If the user's operation is blocked, it will be unblocked as soon as the required data has been copied to disk. This allows the user significantly faster access to her data.

Excluding the volume mount time, the time to access a byte of data being restored for other systems is given by the following expression:

$$TIME_{Byte\ N} = \frac{SIZE_{FILE}}{\min\left[RATE_{TAPE}, RATE_{DISK}\right]} + \frac{Byte\ N}{RATE_{DISK}}$$

Where *SIZE* is the size of the file being restored and *RATE* is the transfer rate of the device specified.

The minimum function expresses that data moves between two devices at the rate of the slowest device.

Again excluding the volume mount time, for RASH, the time to access the same byte is given by

$$TIME_{Byte\ N} = \frac{Byte\ N}{\min\left(RATE_{TAPE}, RATE_{DISK}\right)}$$

The effects of the rapid access feature can be seen in the following example. Given a file size of 200 MB, a tape transfer rate of 1 MB/s, and a typical non-striped disk transfer rate of 300 KB/s; the time to access the first, middle, and last byte is given in the following table.

Access Time (seconds)		
Byte	Other	RASH
0	682	0
100M	1023	100
200M	1364	200

In this example, RASH is clearly superior. This result is biased by the poor performance of the disk file system. A different result is seen if the disk rate is improved to the MSS-II target of 10 MB/s.

Access Time (seconds)		
Byte	Other	RASH
0	200	0
100M	210	100
200M	220	200

While the time to the last byte of the file is only improved by 9%, the time to access the first byte is still effectively zero. To quote Dave Tweten, the NASA project leader: *"At times, the illusion of performance is as important as real performance."*

3.2.2. RASH Buffer Reuse

There is another major benefit that is a side effect of the "block on access" feature. The "other" systems will read blocks of data from the archival storage and write them to disk. After the whole file has been copied, the user is allowed to access the data. Except for short files, when the user accesses the data, it will have been flushed from the system buffers and must be reread from disk. When RASH reads a block of data from the higher storage level and writes it to disk, the user's read or write operation on that data is unblocked. The read or write operation will likely find the data already in a system buffer. This saves a disk access and provides even faster access to the data.

3.2.3. RASH Inode Flags

Another major difference from BUMP stems from the fact that the inode was modified. RASH maintains in the inode a set of status flags and assigns each file under its control a *bit file id*. This bit file id, a concept taken from the IEEE Mass Storage Reference Model,[MILLER87] is always unique. Since the RASH status flags and bit file id are never overwritten as is the handle in BUMP, the file is always connected to its database records. Should an archived file be restored and then again become a candidate for archival, the status flags (and bit file id) still exist and no action need be taken other than the release of the disk space. In a similar situation, BUMP must again copy the file to archive media because the file handle has be overlaid by the disk block addresses.

3.3. The Archival Process

Archiving is a multiple step process. The initial process runs nightly via *crontab* to maintain sufficient free disk space on a specified file systems. This process, named **rasharch**, reads a list of file systems and the percentage of free space to maintain on each. Rasharch forks a process, **archbuild**, for each file system.

Archbuild walks the file tree starting at the file system root and produces a list of every disk resident regular file on the file system. The list consists of the file name, uid of owner, the size, and a selection value. The selection value is currently "time since last access". It may be changed to whatever algorithm the systems administrator wishes, such as a time×space product. The file list is sorted by the selection value. This file list is now available for the next step in the process.

Upon completion of archbuild, rasharch forks another process, **archive**, per file system. "Archive" may also be started as a result of a "panic archive" request from the kernel. In either case, it uses the same file list built by archbuild. "Archive" determines the amount of space which must be reclaimed, selects sufficient files from the existing file selection list and removes those files from the list. "Archive" then sort the selected list into "large" and "small" files. The "small" file list is then resorted by user. Each group, "large" files and per user "small" files, will be handed to a **doarc** process.

Doarc processes a group of files. If the file has already been archived, it will not be copied again. Otherwise, a large file is written to its own set of volumes. Small files are added to the user's currently active small file volume. As the files are copied to volumes, the various database relations are updated.

After the file has been successfully copied, the inode is updated to reflect its "archived" status. If no other process has the file open, the disk space is released.

3.3.1. The Restoration Process

The restoration process starts when the RASH daemon, *rashd*, receives a restore message from the kernel. Rashd forks a copy of itself to deal with the restore and the parent returns to monitoring additional requests.

The sequence of events in restoring a file is:

1. *Rashopen* the inode.
2. Validate the restore request and check the database.
3. Fork one or more children to obtain the off-line volumes and copy the file using *rashwrite*. Update the current resident size with a *rashcntl* function. For large files on several volumes, the copy from each volume may be in parallel.
4. If an error occurs on reading from a volume, switch to the backup volume.
5. If an fatal error occurs, set the RRFAIL flag.
6. *Rashclose* the file.

The RASH daemon also handles the "obsolete" and "panic archive" requests. For obsolete requests, rashd forks a copy of itself to update the database. For panic archive requests, rashd starts the archive process at the **archive** task with a flag indicating it is a panic run.

3.4. The RASH Database

RASH is written using a commercial relational database. Various tables or relations are defined to contain information about the archived files, the removable volumes, and general media.

- The *uservol* relation contains owner and VSN information regarding each user's active *short file* volumes. These are the volumes to which short files will be written. If there is insufficient space to hold additional short files, these volumes are closed and new volumes are assigned. There are two active volumes per user, a primary volume and a backup volume.
- The *fileinfo* relation contains information about each file archived. The bit file id is the key into this relation. The file name, user id and name, group id and name along with the file time stamps are maintained here. A count of number of volumes in the set holding the file is included. A flag is set when an archived file is modified or removed. The entry is then kept until a specified date, when it removed from the database.
- The *filevsn* relation maps each file to the volume or volumes on which it resides. There are two records per volume for each file, a primary set and a backup set. The file's position on a multi-file

volume or the sequence of the volume in a multi-volume set is included. A physical location on the media for the file header label is maintained for rapid "seeking". For volumes which are in a multi-volume set, the offset into the file for the first byte of the file which is on this volume is also maintained (useful for parallel restorations).

- The *volstat* relation provides information about each volume under control of RASH. The physical location of the first available byte on the media, time stamps of volume usage, the uid and gid of the owner of the files on the volume, and the amount of available space on the volume are maintained. A fail-flag is set if an error is detected on the volume; the volume will be discarded and not reused as soon all existing files become obsolete.
- The relation *media* provides information about the different types of media supported by RASH. The maximum size of a piece of a large file which will fit on this type of media or the approximate size of the sum of small files which will fit is specified. The cut over size that determines if a file is large or small is also specified here.
- One additional relation, *level*, describes the limits of the media in each level. For the media type, the current and maximum number of volumes allowed in the level is specified. The current and minimum number of free volumes in this level is maintained. This relation drives the migration of volumes from one level to another.

3.5. UNIX Kernel Modifications

The modifications to the UNIX kernel were kept as simple and isolated as possible with the rapid access feature in mind. The modifications fall into four groups: a kernel←→daemon message driver, changes to the inode structures, new RASH functions, and modifications to existing kernel functions.

3.5.1. Kdcom - a Kernel to Daemon Message Driver

Three events detected by the kernel require action by the RASH user processes, opening an archived file, modification or deletion of an archived file, and low disk space. This is accomplished by having the kernel pass a message to a daemon which either takes direct action or forks another process to perform the required action.

For the message passing, a general purpose, two way, kernel←→daemon message passing driver was developed. This driver, called *kdcom*, for kernel to daemon communication, provides for a daemon to either read a message from the kernel or to write a message to the kernel, though RASH only reads messages from the kernel and does not write back. The driver is written as a character device driver with all normal interfaces: open, read, write, close, and ioctl. Two special kernel entry points, kdcput and kdcget, allow for the kernel to write and read messages and correspond to the interrupt level routines of actual device drivers. Although written as a character device driver, kdcom is intended to pass fixed length messages.

Kdcom will support any number of minor devices; multiple daemons may concurrently use kdcom, each on its own minor device. Within kernel memory, a pool of message buffers is reserved, the number of which is definable at system generation time. The message buffers are linked into one of several lists: the free list, an outgoing list for each minor device, or an incoming list for each minor device.

3.5.2. Changes to the Inode

The following fields are added to the on-disk inode structure:

short	di_hlevel	storage level
long	di_rashflg	RASH control flags
long long	di_bitfid	Bit File ID (long long is 64 bits)

In addition, the following additional field as been added to the in-core inode:

off_t	i_resisize	the current resident file size

The rashflg field has the following flags defined:

RARCH
The file has been archived, i.e. copies exist on another storage level.

RNONR
The file is non resident on disk; the disk space has been released and it must be restored to disk from a higher level of storage before usage.

RRIP The restore operation is in progress for this file. Additional opens will not start another restore and accesses past the current resident size will be blocked.

RRFAIL
The restore operation failed for this file. The user can access whatever data was successfully restored, but attempts to access past the current resident size will result in an **ERESTOR** error. Since the file remains in the non-resident state, another open will retry the restoration of the file.

RAIP An archive of this file is in progress. This is mainly a flag between the various archive processes to indicate success or failure. Additionally, a user write on the file will clear this flag indicating that the archive copy is obsolete.

RWANT
A user process is sleeping on a change to the RASH status of the file; typically this means a read or write past the current resident size. When this field is updated or the restore completes, any process sleeping on this event will be awoken.

3.5.3. RASH Kernel Code

The new kernel functions required to support RASH are packaged into a single module. The following is a brief description of each of the RASH kernel functions. Rashopen, rashclose, rashwrite, and rashcntl are used by the restore process when an archived file is opened. Rashcntl is also used by the archive process and ls(1).

Rashopen
corresponds very closely to the normal system open(2) call. One major problem requires that a different system call exist for the restore process. The normal open call takes a path name, and passes it to *namei()* to locate the inode. If the path name is relative to the user process's current working directory, it cannot easily be made absolute to be passed to the RASH daemon. Furthermore, since the inode has already been located and brought into memory, it is simple to pass the *mount pointer* and the *inumber* to the daemon. The "bit file id" is also passed in the restore message. The RASH daemon then "opens" the file with the special rashopen call which uses the mount pointer and inumber rather than a path name.

Rashclose
performs all the functions of the normal system close(2) call on a regular file. If the restore operation was successful, rashclose then clears the restore is in progress flag, clears the current resident size, clears the non-resident flag in the inode, and awakens any process sleeping on the RASH flags.

Rashwrite
is a special case of the normal write(2) system call. It is a separate routine to bypass the added check in rdwr() which blocks the write if it is accessing data not yet restored. Since this is exactly what the restore process is doing, a way around the block is required.

Rashcntl
provides several bookkeeping functions for the restore and archive daemons. It is also the only RASH system function which is callable by a non-super user process, but only for the get-status functions. Rashcntl provides the following capabilities as commands:

RESET –
Only used when the RASH daemon first starts, kills any restore operation which might have been in progress. This operation scans the in-core inodes and sets RRFAIL in each inode

where RRIP was set. This prevents files hanging in "restore mode" should the system crash.

PARCD –
Tell the kernel that the request "panic archive" run has been completed.

GETST –
Returns a *rashstat* structure to the calling process for the file pointed to by the file descriptor. The rashstat structure contains the fields added for RASH.

GETPST –
The same as GETST except that a path name is used rather than a file descriptor. The *ls(1)* command uses GETPST rather than GETST so that the file is not opened.

SETARC –
Used by the archive process to mark the file as archived. If either the RRIP or RARCH flag is set, set the RARCH flag. If no other process has the file open, release the disk blocks assigned to the file, clear the disk address array in the inode and set the non-resident flag, RNONR.

SETFLG –
Used by the archive process to set various flags such as RRFAIL.

SETBID –
Insert the assigned bit file id into the inode.

SETRSIZE –
Update the current resident size field in the inode and awaken any process sleeping on the rashflg field.

SETLVL –
Used to update the RASH storage level field in the inode when the file is moved to a different storage level.

Call_rashd
is used by the kernel to format a message to the RASH daemon. Called with the command (restore or mark obsolete), and a pointer to the inode, call_rashd will extract the *mount pointer*, *inumber*, and *bit file id* from the inode, format the message and call *kdcput* to send the message. The messages have the following structure:

```
struct rashcall {
        int     id;             /* a sequence number for tracking */
        int     cmd;            /* the command to the daemon */
        struct mount *mp;       /* mount pointer for the file system */
        ino_t inumber;          /* the inumber */
        long long bitfid;               /* the bit file id */
}
```

Rash_panic
is called when *alloc()* detects that the free space in a file system reaches a critically low level. When called with a mount structure pointer, rash_panic will send a message to the RASH daemon which will fork a process to archive files. The RASH daemon is used as the receiver as it is always listening so a second daemon is not required. To prevent additional panic archive messages on the same system, an array exists in the kernel which has one entry per kernel mount structure. The element which corresponds to the full file system is incremented. Only on the transition from zero is the message passed to kdcom. When the panic archive completes, the panic archive process will issue a rashcntl function, R_PARCD, to reset the array element to zero.

Rash_delay
is used by kernel routines, such a *rdwr()* to block until the data being accessed is available. If the restore in progress flag is set and if

u.u_offset + u.u_count > i.i_resisize

then the process is put to sleep with the flag, RWANT, set. Upon wake up, if the "restore failed" flag, RRFAIL, is set; u.u_error is set to ERESTOR.

3.5.4. Modifications to Existing Kernel Functions

In addition to the new RASH functions added to the kernel, ten existing kernel functions in five modules have had code added.

Alloc()
calls *rash_panic* to request a panic archive run if the total number of free blocks drops below a limit (currently 3.125%, or a right shift of 5).

Ialloc()
clears the new RASH fields in the inode when it is allocated.

Iread()
copies the new RASH fields into the in-core inode from the disk inode.

Iupdat()
copies rash fields from in-core inode to disk inode.

Iput() sends an obsolete message via call_rash() if the inode link count goes to zero and the inode is being freed.

Core() clears i_rashflg and sends an obsolete message via call_rash if the "core" file has been archived. This is required because core() does not open the file but calls writei() directly.

Gethead()
requests a restore if the file is archived because exec() does not open the text file. It also sleeps if RRIP is set. This is the one case where access to a file is blocked until the whole file has been restored. It is done because gethead() checks to see if anyone has the file open for write, i.e. the restore process. If so, u.u_error is set to ETXTBSY.

rdwr() calls rash_delay() if a restore is in progress. This will delay the access until the data being accessed is available (based on the current resident size, see the discussion of rash_delay above). If the operation is a write and the file is archived, rdwr() requests that the archive copy be marked obsolete. Also on a write, if an archive is in progress (RAIP is set) then it is cleared to indicate the file has changed and the archive copy is bad.

Copen()
if the file is being opened with *O_TRUNC*, (1) delays if a restore is in progress so all blocks will be released, and (2) if the file is archived, requests the copy be marked obsolete. If the open is not done with truncate and the file is not resident and not being restored, a restore request is issued via call_rash().

Close()
will delay until a restore in progress completes if the open mode was write. This is to insure that the user's file is safely on disk when the close completes, part of the MSS-II safe file system feature.

4. The High Performance File System

The largest single barrier to the use of UNIX in an I/O intensive environment is its rather dismal performance in the area of file system I/O. This area has changed little over the years. The only common practice among those who implement UNIX has been to increase performance by enlarging the file system block size[McKusi84] or by increasing the memory buffer cache[Smith85].

Some solutions have included specialty file systems, such as the Contiguous File System[Ampex79]. These have provided some increase in file system performance but not without major drawbacks such as required preallocation and no file growth. The amount of performance gain in these systems is still limited by the performance of a single disk drive and controller combination.

This section describes a UNIX mainframe implementation of a High Performance File System (HPFS) which is a major part of MSS-II. The file system employs disk striping, volume seek ahead, and full tracking I/O for individual files. The new file system also allows multiple partial stripe I/Os to occur simultaneously, giving a significant fraction of the possible striping performance to randomly accessed

files. The HPFS performs much better for sequentially accessed files, both small and large, and somewhat better for random accesses to large files and about the same as UNIX V5.3 for random access to small files.

File system striping is a recent popular solution to the problem of performance. The concept is simple, spread the contents of a single file across multiple data paths, hence multiple disk drives, and achieve faster transfers by accessing the data in a parallel fashion. This type of file system is almost the opposite tack from a standard UNIX file system where it is common to have multiple file systems on one disk drive. Striping file systems have been commonly seen in the supercomputer industry where files are characterized as being both large and sequentially accessed; traits which can fully utilize striping performance.

The HPFS design marries the striping file system with data redundancy to provide a high performance standard UNIX file system.

4.1. Definition of Terms

A striping file system is one which performs simultaneous multiple device I/O on one file. This has traditionally taken the form of having multiple disk controllers, each controller supporting one or more disk drives. The controllers form independent data paths to and from memory. Portions of the file are spread across the disks on these separate data paths. The number of data paths (controllers) is also referred to as the *stripe width*. The number of disk drives on one data path is called the *stripe depth*. A stripe is the allocation unit which is spread across the stripe width. A *striped block* is a block from one disk on each data path. A *striped track* is a track from one disk on each data path. The striped file system can be viewed as a two dimensional array of disk volumes, where the columns are defined as disks which share the same data path and the rows are disks which have independent data paths (see Figure 2).

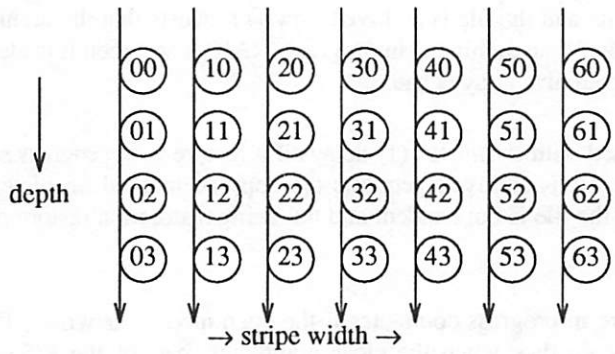


Figure 2: Striping File System Layout

Disk drives or volumes are units which can handle individual seek commands. *Cylinders* are made up of the individual actuator positions on a volume. *Tracks* are the various surfaces within one cylinder. *Blocks* are different rotational positions around each track. Note that for some disks, sectors are directly substitutable for blocks, this document uses blocks throughout.

A *bitmap* is a description of the allocation of space on the file system. One bit contained in a bitmap describes an amount of space on the disk(s). A *bitmap bucket* is a slice of the bitmap used to allocate space for individual files, this is described in detail later.

4.2. Design Overview

The overriding theme of this design is that this new file system should not be radically different than the current implementation. This helps the implementation because a great deal of code might be shared, which means that less code is added. This is also true for data structures such as buffers, superblocks and inodes. Others,[VanBaak87], have taken this approach to designing improvements to UNIX

file systems. More specifically, the HPFS design calls for the use of standard sized blocks, shared use of the memory buffer pool, standard inode and superblock definitions, and even the sharing of the same device driver.

Four important concepts emerge from the design. First is the increase in performance by striping the I/O across multiple data paths, without loss of reliability. Second is the achievement of striping behavior by mapping logical block numbers onto physical blocks; this is critical to the design. Third is the allocation technique which allows for the maximum utilization of the available data path capacity. Fourth is the ability to detect sequential access operations and maximize performance by queuing plenty of read ahead blocks and, at a lower level, optimizing the current queue of requests into the minimum number of I/O operations.

4.3. Performance without loss of reliability

Performance is gained simply by having a number of disks, each with its own data path, participate in the transfer of data for 1 file. This is the basis of a striping file system.

A large problem with striping file systems is one of decreasing reliability with the addition of more disk drives. This problem has been explored by Patterson, et al.[Patter87]. They characterized the general solution as Redundant Arrays of Inexpensive Disks or RAID. They identified 5 members in the redundancy organization taxonomy, each member requiring different hardware and software configurations. They labeled these as level 1 through level 5 RAIDs.

The problem is stated by the following formula:

$$MTTF_{STRIPE} = \frac{MTTF_{DISK}}{\#of DISKS}$$

where MTTF stands for Mean Time To Failure, usually a number provided by the manufacturer of the disk drive. As Patterson showed, the MTTF of a group of 100 disks whose individual MTTF was 30,000 hours (greater than 3 years) is only about 300 hours, or less than 2 weeks.

Briefly stated, the different levels of RAIDs combated the reliability problem by adding redundancy to the disk array. The level 1 RAID is simply a mirrored disk, which is expensive in both data path capacity and in disk space. The level 2 RAID employed a Hamming code error correction technique requiring less overhead resources, only 12% to 38% of the stripe size, but requiring that data be transferred a stripe at a time. This is similar to the memory organization of a computer employing SECDED. Level 3 RAID recognizes that detecting errors is (probably) already built into the drive, so all that is needed is a check disk for the stripe, thus further reducing the overhead cost, but still having the restriction of stripe reads and writes. Level 2 and 3 RAID assume bit or byte interleaving across the stripe. This type of striping is explored by[Kim86] and performs better with synchronized disks. The fourth level RAID assumes block interleaving across the stripe. This allows for independent I/O to occur within the stripe and avoids the necessity of synchronized disks. The partial stripe I/O requires a read/modify/write cycle involving both the data to be replaced and the check disk for writes. The bottleneck is identified as the check disk, since it is involved in every partial stripe write. In the fifth and final level RAID, the check disk information is spread evenly across all members of the stripe. This allows for multiple partial stripe writes to occur simultaneously, although the read/modify/write operation is still required.

Level 1 RAID allows for the most I/O operations per second to occur as there is no check disk (every disk is mirrored). The drawback is the expense of halving the disk and data path capacity. Levels 2, 3, and 4 all perform fairly well (near data path capacity) for large sequential I/O operations, with levels 3 and 4 achieving more available disk capacity by only providing a single check disk string. However, they do not perform well at all on small random I/O operations, achieving only about 10% of available data path capacity. The level 5 RAID performs the same as 3 and 4 for large sequential I/Os, but is much better on small random I/O, achieving 60% of available data path capacity. These calculations were performed for a stripe width of 10 disks, other widths may perform differently.

The probability of failure of the group of redundant disks can be characterized as the probability of a second disk failure before the first failed disk has been repaired or replaced. With the single check disk

on a stripe width of 10 and depth of 1 and a single disk Mean Time To Repair of 1 hour, the MTTF of the RAID is calculated to be 820,000 hours or better than 90 years, with a stripe width of 25 it is over 40 years, clearly exceeding any requirements.

The HPFS design employs a level 5 RAID approach, although the actual implementation is not on "inexpensive" disks. The acronym humorously bestowed on the HPFS implementation is RAVED for Redundant Array of Very Expensive Disks.

4.4. Logical Block Mapping of the HPFS

The mapping of logical blocks to physical blocks is the basis for achieving striping performance for single files while still allowing random block I/O to occur.

Components of the disk physical address form a 3-tuple whose elements are cylinder, track, and block. The traditional method of mapping logical blocks to the physical 3-tuple is to order the 3-tuple to disk[cylinder, track, block]. The reason to map a disk this way is to provide a minimum distance in both rotational delay and seek time to logical blocks which are numerically near each other. By understanding the characteristics of the hardware involved, the ordering might change. This is most often seen as block interleaving, where the hardware is unable to finish a transfer and start another by the time the next rotational block comes under the head.

The HPFS design has 5 components of the physical disk address. These form a 5-tuple whose elements are data path or column, row, cylinder, track, and block. The last three have the same definition as the last example, the data path refers to different disks across the stripe width and the row refers to different disks down the stripe depth. There are now numerous different orderings to the mapping, many of which can provide a minimum time between numerically near logical blocks. Because blocks across the stripe have no delays from one another (simultaneous access), this then is the index which will be the last in the ordering, or vary the fastest. Blocks around the track are chosen for the second to last index. Rows are picked next. This is because the seek time on the next row can be hidden by the track's worth of blocks being transferred on the previous row. Picking the next track on the same disk achieves the same goal but has the detrimental effect of "locking up" a disk for N tracks of data, where N is the number of surfaces. If another I/O is allowed on the same disk before all the tracks are transferred then two seek time penalties are required. By placing the row next in the ordering, the amount of read ahead need only be 2 or 3 stripe tracks worth of data, reducing the amount of system buffer space required and allowing other processes to access data without waiting for N rotations of the disk. The next component in the ordering is the tracks within the cylinder, and the first index in the ordering is therefore the cylinder within the disk. The ordered 5-tuple now looks like diskarray[cylinder, track, row, block, data path].

An Example of logical block mappings, using disks with 4 blocks per track, 4 data paths and 2 rows on each data path (8 disks in the array) is shown in figure 3.

Remember from previous discussion that a level 5 RAID is the design model for the HPFS. This means that one of the blocks in each stripe is the check block. The check block needs to move from data path to data path in sequence in order to avoid bottlenecks on partial stripe updates. If a stripe allocation method is used then there is also a need to spread the starting block in the stripe across the data paths for the same reason. If all files started on data path 0, then small files, those less than a striped block in size, would tend to interfere with one another because they occupy the same data paths. By mapping the logical blocks so that if block N occurs on data path K then block N+stripe_width occurs on data path K+1 modulo stripe_width, both problems can easily be solved. The check block is now the last logical block in each striped block.

Now the logical block mappings are: LB 0 maps to data path 0, row 0, cylinder 0, track 0, block 0; LB 3 (a CHECK block) maps to data path 3, row 0, cylinder 0, track 0, block 0; LB 4 maps to *data path 1*, row 0, cylinder 0, track 0, block 1; LB 7 (a CHECK block) maps to *data path 0*, row 0, cylinder 0, track 0, block 1 (See Figure 4).

By choosing this mapping scheme, only 2 or 3 striped tracks of data need be requested at one time to achieve continuous striping performance. This statement is based on the assumption that file allocation occurs such that major portions of the file have sequential logical blocks and the assumption that the device driver layer can organize multiple I/Os to a disk in an optimal fashion.

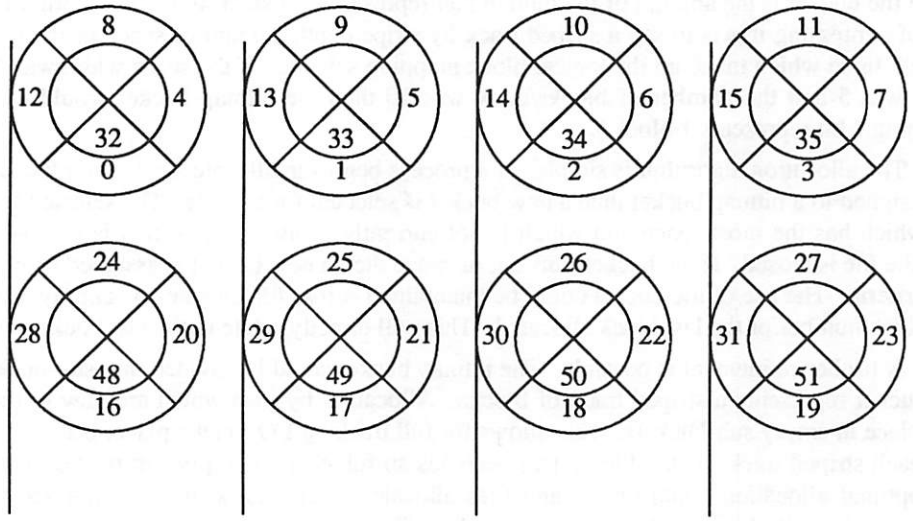


Figure 3: Logical Block Mapping

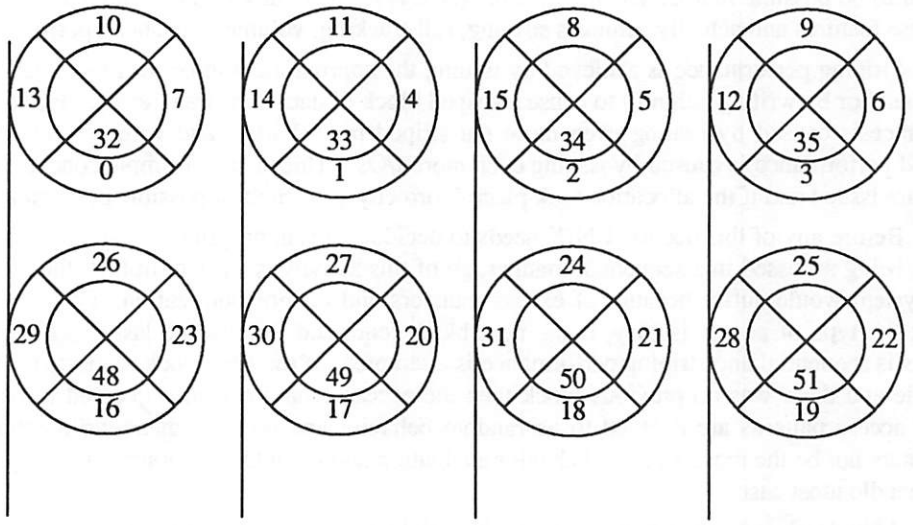


Figure 4: Logical Block Mapping with start block rotation

4.5. New Allocation Scheme

As demonstrated in the last section, striping performance to a single file can only occur if major portions of the file are in sequential logical block order. The current V5.3 scheme of LIFO (Last block In to free list is the First block allocated or Out) allocation is inappropriate for this task. A much more appropriate approach is that taken by BSD 4.2[McKusi84]. This approach breaks the disk into cylinder groups and allocates space for files from a cylinder group bitmap. The bitmap contains a bit for each block of data on the disk, it is marked as allocated by a 1 and free by a 0. If the file system needs N blocks sequentially ordered, it need only find a group of N 0 bits in the bitmap. It is much faster to scan the bitmap for a group of free blocks than it is to search a freelist.

The HPFS design does not need cylinder groups as defined in BSD 4.2, it does need a similar idea to cylinder groups, however, in order to continue allocation for a given file in the same striped track

where it last allocated a block. In the HPFS design, the cylinder group is known as a bitmap bucket, where the bucket is the amount of the bitmap that represents a disk array track's worth of space. Another way of expressing this is to say a striped track by stripe depth amount of space or the last 3 components of the 5-tuple which make up the logical block mapping scheme. If the stripe width was 10 and the stripe depth was 5 and the number of blocks/track was 10 then the bitmap bucket would be 500 bits long (assuming 1 bit represents 1 block).

The allocation algorithm is simple. If a process begins to allocate space for a file and it currently is not assigned to a bitmap bucket then a new bucket is selected for this file. The selected bucket will be the one which has the most space and which is not currently in use. The bucket is marked as being in use until the file is closed. If the bucket runs out of space then a new bucket is assigned using the same selection criteria. The use of the bucket could be maintained across file closings by simply examining the logical block number of the last block allocated. This will directly relate to the last bucket used by this file.

A further refinement is possible. The bitmap bucket could be divided into sub-buckets, where each sub-bucket represents a striped track of blocks. Allocation by files which are new to the bucket should take place in empty sub-buckets. This allows for full tracking I/O to take place, because only 1 file occupies each striped track. If the file system becomes so full that no empty sub-buckets can be found, then non optimal allocation could occur, and files allocated after this occurs will not see the full tracking behavior, although they could still see striping behavior.

4.6. Striping Performance in a UNIX Environment

With the logical block mapping scheme and the bitmap allocation scheme, striping performance is shown to be possible in a UNIX file system. What is left is to describe how the HPFS design makes use of these features and actually achieves striping, full-tracking, volume seek ahead performance.

Striping performance is achieved by issuing the appropriate number of block I/Os (either by reading ahead or by writing behind) to cause a striped block of data to be transferred. Full track, striped performance is caused by issuing even more (a striped track) I/Os; and volume seek ahead, full track, striped performance is caused by issuing even more I/Os. This then is a simple concept, if enough block I/Os are issued and if the allocation took place "correctly", the highest possible performance is achieved.

Before any of this occurs, UNIX needs to decide if it is appropriate to even attempt this. If the file is not being accessed in a sequential manner, all of this activity is wasted. In fact, the performance of the file system would suffer because of excess transfers and memory utilization. Fortunately, the decision about the type of access is easy, if the next block requested follows the last block requested, then the access is sequential and striping performance is attempted. If the next block requested is the first block in the file and there was no previous block then the access behavior is also deemed to be sequential. All other access patterns are deemed to be random behavior and no read ahead/write behind is attempted. This may not be the most accurate behavior evaluation and it could occasionally miss, but it is simple and will handle most cases.

After having decided that this is a sequential access and issuing the appropriate number of I/Os, how does striping occur? The Block I/O layer in UNIX queues the I/O to individual device queues, the device driver examines each devices' queue and, if the device is not busy, issues the I/O. Because of the logical block mapping and the allocation scheme already discussed, striping simply happens. Full tracking is another matter; the device driver will have to be modified to examine the device queue and make the determination that a full track I/O is possible. Full tracking is possible if all of the blocks which make up a single track on this disk appear on the I/O queue and, furthermore, they are all to be read or written (intermixing is not allowed). If these conditions are met then the driver can construct a single I/O to transfer the entire track, making use of known rotational position or, in some cases, simply ignoring the rotational position (soft formatting the track as it is being written). Volume seek ahead occurs as automatically as striping, i.e. if enough I/O is queued to the devices then the device driver will issue the I/O to both the current row and to the next row (next in the stripe depth). These disks can seek to right actuator position but they cannot start transferring until the previous row is finished using the data paths. As the current row finishes the next row can start transferring and a whole new set of read ahead / write behind blocks are queued to the yet another row of disks in the stripe depth.

By changing the file I/O layer in UNIX to recognize sequential access patterns, and by issuing the correct amount of read ahead / write behind block I/Os, and by changing the device driver to recognize full track I/O opportunities, the HPFS design achieves striping, full tracking, volume seek ahead performance.

4.7. Calculation and Placement of the Redundancy Information

As mentioned in section 2.2, each stripe has a check block associated with it. The check block rotates from disk to disk in order to prevent partial stripe updates from interfering with one another. The logical block mapping scheme takes care of this detail, providing a uniform look to the placement of the check block. The check block is always the largest logical block number in the stripe.

The check blocks cannot be allocated to any file, they must appear as "preallocated" space in the bitmap. As such, they are never scheduled for I/O operations in the normal sense.

The check blocks are not needed at all during normal read operations, full or partial stripe. However, during a full stripe write operation, the check block needs to be calculated and written. Even worse, during partial stripe updates, the check block *and the previous data block* need to be read and a new check block calculated and written. All of this activity occurs when UNIX has decided to reclaim a system buffer or flush dirty buffers.

The calculation of the contents of the check block is extremely simple. The blocks which make up the stripe are logically differenced (exclusive or) with one another in much the same way a parity bit is calculated. If a disk has failed, the contents of the missing block can be reconstructed by performing the logical difference calculation on the remaining blocks, including the check block. If it is the check block which is missing, then when the repair is finished, the check block could be reconstructed. If the "keep it simple, stupid" approach is taken here, then the new disk would have the product of the logical sum operation of all the other disks in the row. This replaces both the missing data blocks and the missing check blocks contained on the failed drive.

4.8. Block I/O to Device Driver Communication

As mentioned previously, the block I/O layer needs to queue a large number of read ahead or write behind buffers to the device driver. This could be accomplished by making repeated calls to the routines which assign buffers and to the device driver strategy routine. This would follow the normal way that UNIX works. This approach has a few drawbacks; the device driver could start a transfer prematurely, before all of the blocks which make up a track are queued, thus eliminating the full track I/O opportunity. Another drawback is the shear amount of overhead involved in making repeated subroutine calls.

A better method is needed for the HPFS design. The HPFS design gathers a group of buffers from the freelist and queues them *en masse* to the device driver. This can be accomplished by making a few changes to the standard UNIX code. The buffers to be passed to the device driver are linked together using the av_forw and av_back pointers in the buffer header. These pointers are not used after the buffer has been removed from the freelist, although some implementations use these pointers in the device driver to form the queue of current I/O requests. In any case, they are not used between the time of freelist removal and the I/O queuing, so they can be used to form the chain of requests to the device driver layer. The actual call to the strategy routine remains the same. Other routines which pass buffers to the strategy routine will have to be changed to terminate the chain with the first buffer. The device driver will need to be changed to follow the chain of requests, queuing the buffers to the appropriate drives' I/O queue. After the last buffer is queued, then the non busy drives can have I/O started on them.

4.9. Buffer Management and the Check Block

There is a new concept which the buffer management layer needs to address. This is the concept of requiring a free buffer in order to actually write a dirty buffer to disk. The HPFS design could require this free buffer for either of two reasons, either because a check block needs to be calculated or the previous contents of a block are needed for a partial stripe update.

UNIX keeps all buffers in the buffer pool on one of two disjoint lists. The two lists are the freelist and the awaiting I/O list. A buffer may only exist on one or the other of these lists. In addition, the buffer may be placed on a hash queue list. This list is used to speed the process of finding a cached block. Buffers may exist on the hash queue list and one of either the freelist or I/O list simultaneously. Various flags are marked in the buffer header to indicate the state of the buffer.

A process doing I/O will first attempt to find the required block in memory, if the block cannot be found then a buffer is requested from the freelist. If the freelist contains buffers marked for delayed write, the process schedules the buffer for I/O and continues searching. If the end of the freelist is reached, the process sleeps. The process will be awoken when a delayed write is finished and the buffer is returned to the freelist. HPFS has a problem with this scheme. If all of the buffers on the freelist are marked for delayed write, a deadlock situation occurs. The delayed write cannot start until a free buffer is available for either a check block or for a previous data block in a partial stripe update. If all of the buffers are on the freelist, then no new buffers will be placed on the freelist by I/O completion and no buffers will ever be available for the check block calculation.

The solution to the problem is to reserve a certain number of buffers in the pool which cannot contain delayed write blocks. This is implemented by counting the buffers in the pool which do not have the delayed write status. Every time a delayed write occurs, the count is decremented. If the count reaches a known fixed value, a stripe write of delayed buffers is started. As these I/Os start, the count is incremented. This is similar to what happens now in UNIX, except the count is implied to be zero by the code. The reserved value is determined as the minimum number of buffers needed to do at least one partial stripe update. For the worst case partial stripe write this value is the stripe width minus one plus one for the check block for this stripe. If the activity of the file system is such that delayed writes are sleeping waiting for the first few stripe writes to complete, then the number of reserved buffers should be increased. Note that these buffers are not reserved empty, they may still contain cached read data.

A second change is required for partial stripe updates. HPFS needs the old contents of a particular disk block in order to remove its contribution to the existing check block, before adding the contribution of the replacement block. This implies that two buffers will exist in memory for the same disk block, the old and the new. A flag is needed in the buffer header to indicate that this situation has occurred to prevent accidental cache hits on the old block. Remarkably, a flag already exists in UNIX for a similar purpose, B_STALE. It implies a slightly different meaning however, indicating that an I/O error occurred for this buffer at some previous time. Using this flag for a new purpose may not be the proper way to address the issue.

5. Standardization Issues

In this last section, issues of standardizing a file archive system are addressed.

5.1. Uniform Kernel Hooks

The choice of a "normal" or a RASH approach to archive/restore will greatly effect the nature of kernel "hooks" required for a standard system. RASH requires more hooks than does a system like BUMP. However, even the hooks for RASH are few and straightforward. Both systems require the kernel to notify a daemon upon open of an archived file. Both system must deal with archived file entries in a database when the file is deleted or modified.

The majority of the work in creating an archive system is not in the kernel but in the application code that deals with the requests and the database. It is true that most sites, with a source license, can make the required set of changes. However, having a standard, vendor supported set of hooks in the kernel will lessen the development effort and eliminate another source of local kernel maintenance.

5.2. Inode Changes

The changes to the disk inode prove to be the most complex change for a site to make. Even with *vnodes* or the *file system switch*, a modification which involves changing the size of the inode is an enormous undertaking, being pervasive throughout the file system utilities. Yet, as UNIX and

"supercomputing" merge, we see a growing need for major changes to the file system and inode structure.

The file size limit alone becomes a problem for many users of supercomputers. Two gigabytes is becoming a serious limit for many users. With the advent of striped or multi-volume file systems, the size or offset field (in the inode, user area, and stat structures) becomes the limiting feature.

Implementation of a full featured archive system, like RASH, also requires additional fields in the inode. BUMP, the Cray data migration project, and other like them are fine as far as they go. However, they are restricted by the inability to store information about the archived file in its real inode. Based on the number of sites and companies which are addressing the file archival problem, accounting, and system security issues; a strong need for additional, site usable, space in the inode and other structures is called for.

5.3. Vendor Challenge

This paper has addressed much of the effort NASA has put forth to improve the performance and capability of UNIX. Both the BUMP and RASH projects encountered many of the same problems in adding archival capability to UNIX; RASH was willing and able to modify more of the system than was BUMP. The time has come for features such as archival systems to be standardized and supported by the vendors.

Therefore, this paper ends with a challenge; a challenge to the UNIX vendors and standard committees, to the System V camp and the BSD camp. If UNIX is to continue to grow in capability and performance, if new features such as file archiving are to continue to flourish, the cooperation of the UNIX vendor community is required.

Do not be bound by outmoded concepts or old hardware limitations. UNIX is the environment and function; UNIX is not the old code. Be willing to be different. Be free to add features, add capability, add performance, add space in structures. Provide space for the sites to use in the disk inode and other critical tables which are difficult to change. Help, do not hinder your customers. Let UNIX grow.

6. Acknowledgements

Appreciation is expressed to the other members of the project team for their efforts upon the project and their support for this paper; they are Dave Tweten and Mark Tangney of NASA; Tom Proett and Ruth Iverson of GE; Jonathan Hahn of Unetix and Bill Ross of Sterling Software.

A slightly different version of the high performance file system section was presented as a paper by Alan Poston at the Usenix "Workshop on UNIX and Supercomputers" held in Pittsburgh in September, 1988.

References

[NAS88].
Ames Research Center, *Numerical Aerodynamic Simulation (NAS) Annual Report*, 1988.

[CFS88].
Bill Collins and Catherine Mexal, ''The Los Alamos Common File System,'' *Tutorial Notes, Ninth IEEE Symposium on Mass Storage Systems*, pp. 171-180, IEEE, October 31, 1988.

[BUMP88].
Michale Muuss, Terry Slatery, and Donald Merritt, ''BUMP, The BRL/USNA Migration Project,'' *Proceedings, Workshop on Unix and Supercomputers*, pp. 183-214, Usenix, September 26, 1988.

[MILLER87].
Stephen W. Miller, *IEEE Reference Model for Mass Storage*, SRI International, Menlo Park, CA, 1987.

[McKusi84].
M. K. McKusisk, W. N. Joy, S. J. Leffler, and R. S. Fabry, ''A Fast File System for UNIX,'' *ACM*

Transactions on Computer Systems, vol. 2(3), pp. 181-197, August 1984.

[Smith85].
Alan Jay Smith, "Disk Cache - Miss Ratio Analysis and Design Consideration," *ACM Transactions on Computer Systems*, vol. 3, no. 3, pp. 161-203, August 1985.

[Ampex79].
Ampex Corporation, *Contiguous File System*, 1979. Internal Design Document

[VanBaak87].
Thomas VanBaak, "Virtual Disks: A New Approach to Disk Configuration," *Winter 1987 USENIX Association Conference Proceedings*, Washington, D.C., 1987.

[Patter87].
D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks," *Report No. UCB/CSD 87/391 Dept. of Electrical Engineering and Computer Science*, University of California, Berkeley, 1987.

[Kim86].
Michelle Y. Kim, "Sychronized Disk Interleaving," *IEEE Transactions on Computers*, vol. C-35, no. 11, November 1986.

MULTIPROCESSOR ASPECTS OF THE DG/UX KERNEL

Michael H. Kelley
Data General Corporation
Research Triangle Park, North Carolina 27709
kelleymh@dg-rtp.dg.com or uunet!dg-rtp.dg.com!kelleymh

Abstract: This paper describes the multiprocessor aspects of the DG/UX™ 4.00 kernel, a symmetric multiprocessor UNIX* kernel that runs on Data General MV/family computers. In revision 4.00, the DG/UX kernel has been completely rewritten to support multiprocessor hardware in a fully symmetric fashion and to provide a number of other functional and reliability enhancements. The paper focuses on process management, device I/O, and virtual memory management because these areas of the kernel have specific multiprocessor components. A virtual processor model is presented as a method of achieving multiprocessor operation while hiding the actual number of physical processors from the rest of the kernel. The application of this model to process sychronization and scheduling is discussed in detail. Multiprocessor device I/O issues are discussed next and a flexible device driver paradigm is sketched. Finally, multiprocessor issues in virtual memory management are presented with the solutions used in the new DG/UX kernel.

1. Background

The new DG/UX kernel runs on tightly-coupled symmetric hardware. "Tightly-coupled" means that the processors share main memory and run one copy of the operating system. Thus, they appear to the users as one system with the actual number of processors invisible except for system performance. "Symmetric" means that all processors in the system have exactly the same capabilities. They can execute the same instruction set, can access the same main memory, can access all I/O devices, and can handle interrupts [Ensl77].

Operating systems for tightly coupled multiprocessors generally fall into one of two categories: master/slave and symmetric. In a master/slave system, the kernel is limited to running on only one of the processors, which is designated the master processor. The other processors in the system are designated as slaves. Application level execution may occur on any of the processors, but if an application on a slave requests a kernel service, the application must be descheduled and queued for service on the master [Gobl82]. By contrast, in a symmetric system all processors are equivalent and interchangeable. The kernel can execute on any of the processors and may be executing simultaneously on several processors. If an application requests kernel service, the service can be performed without changing processors [Bach84, Hami88].

---

* UNIX is a trademark of AT&T.

The master/slave and symmetric categories are, of course, not absolute. Many multiprocessor systems are hybrids in that some of the kernel can execute on any processor while other parts are restricted only to the master [Arno74, Holl82, Nogu75]. For example, only one of the processors may have access to I/O devices, and hence the device driver portion of the kernel would be restricted to executing on that processor.

The DG/UX kernel was designed as a fully symmetric kernel because symmetric kernels achieve better performance as the number of processors and the amount of kernel execution increases. In a master/slave system only 2 to 3 processors can be used effectively before significant performance degradation occurs due to centralized kernel execution. DG/UX should run well on as many as 6 to 8 processors, and if the load is skewed toward application execution and away from kernel execution, even more processors could be used effectively.

2. Process Management

The multiprocessor nature of the DG/UX kernel affects the synchronization and scheduling aspects of process management. The virtual processor abstraction, originally described in [Reed76], is introduced to provide synchronization and scheduling in a way that is independent of the actual number of physical processors. The following sections describe how virtual processors have been adapted and how the synchronization and scheduling mechanisms have been built to complement it.

2.1 Virtual processors

A *virtual processor* (VP) is an abstraction of a physical processor. The VP abstraction allows the uniprocessor or multiprocessor nature of the underlying hardware to be hidden from the higher levels of the kernel.

An instance of the kernel has a fixed number of VPs that is more than the number of physical processors but usually less than the number of processes wanting to execute. Thus, a two level scheduling scheme is used (see Figure 1). The lower level of scheduling multiplexes VPs onto physical processors so that the VPs appear to be active entities that execute code. This short term scheduling is performed by the dispatcher. The higher level of scheduling multiplexes processes onto VPs so that the processes may execute. This scheduling is performed by the medium term scheduler using the operations defined on VPs. Both types of scheduling are described in greater detail later in this section.

2.1.1 VP state transitions

Figure 2 shows the state transitions a VP can make. The boxes represent the states, while the arrows represent the operations that cause a transition. A description of the states and operations follows.

2.1.2 Bind and unbind

The *bind* operation attaches a process to a VP that is in the unbound state and puts the VP in the stopped state. The *unbind* operation detaches a process from a VP in the stopped state and returns that VP to the unbound state. At most one process may be bound to a VP at any point in time.

Bind also moves information about the process into the more restricted VP environment by copying information into per-VP databases and by making some per-process data structures resident in memory. *Unbind* reverses these effects. Hence, when a process is bound to a VP it consumes more system resources and is in a

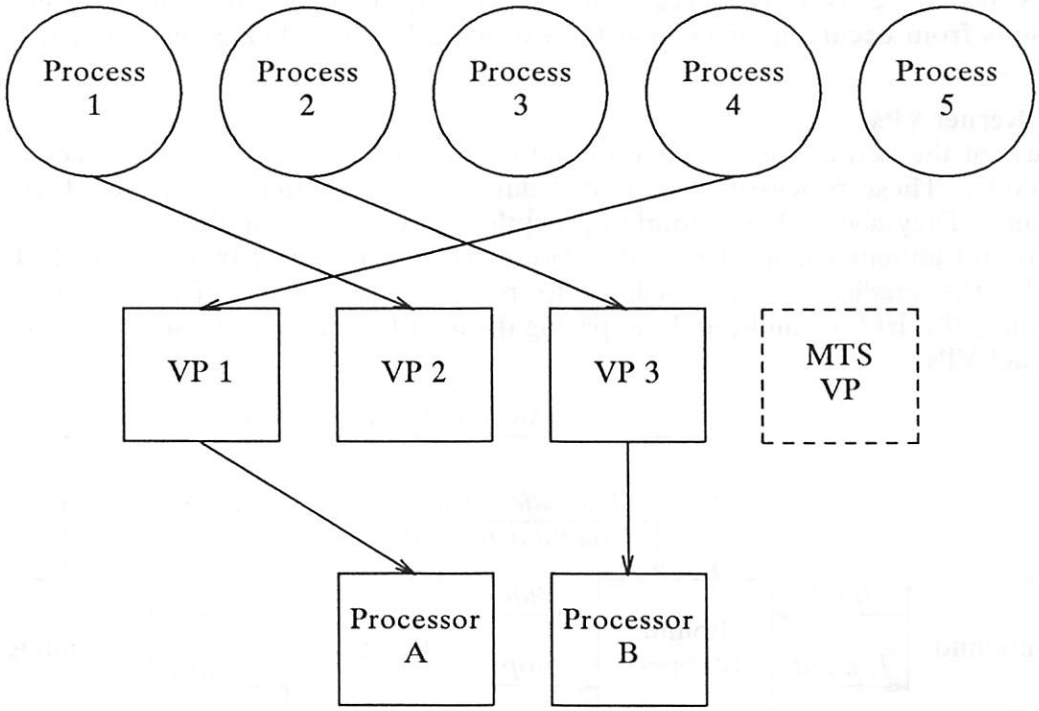


Figure 1. Two level scheduling

higher state of readiness for execution than when it is not bound.

The medium term scheduler uses *bind* and *unbind* to multiplex processes onto VPs.

2.1.3 Run and stop

The *run* operation moves a VP from the stopped state to the running state where it will compete for a processor with the other VPs in the running state. The *stop* operation moves a VP from the running state to the stopped state. *Stop* is asynchronous in that the call may return before the VP actually stops; when the VP does stop, it will cease to compete for a processor. The asynchrony is necessary because the VP may be executing code that must be treated as an atomic operation; *stop* can take effect only when the atomic operation is completed.

In addition to the explicit *stop* operation, a VP moves to the stopped state implicitly when its await quantum or execution time quantum expires. These values are specified when the VP is run and control the scheduling of the VP as described later in this section.

The medium term scheduler uses *run* and *stop* in conjunction with *bind* and *unbind* to regulate overall system load.

2.1.4 Begin and end atomic operation

The operations *begin atomic op* and *end atomic op* are used in sections of kernel code that must execute atomically with respect to *stop* operations. The VP management code maintains an atomic op depth (AOD) so that these calls may be nested, with a non-zero value indicating that atomic operations are in progress.

Non-zero AOD is used to ensure that a process does not *stop* while holding a lock required before it can be *run* again, because otherwise deadlock would result. Non-

zero AOD also effectively increases a VPs priority, though it does not prevent I/O interrupts from occurring or prevent the executing VP from being descheduled by the dispatcher.

2.1.5 Kernel VPs

Parts of the kernel are designed around daemon processes that never execute user level code. These processes are created during system initialization, bound to VPs, and run. They always have atomic op depth of 1 or greater and hence do not stop and are not unbound from their VPs. These VPs are removed from the pool of VPs used by the medium term scheduler to run user processes. The medium term scheduler, the I/O daemon, and the paging daemon (all described later) are examples of kernel VPs.

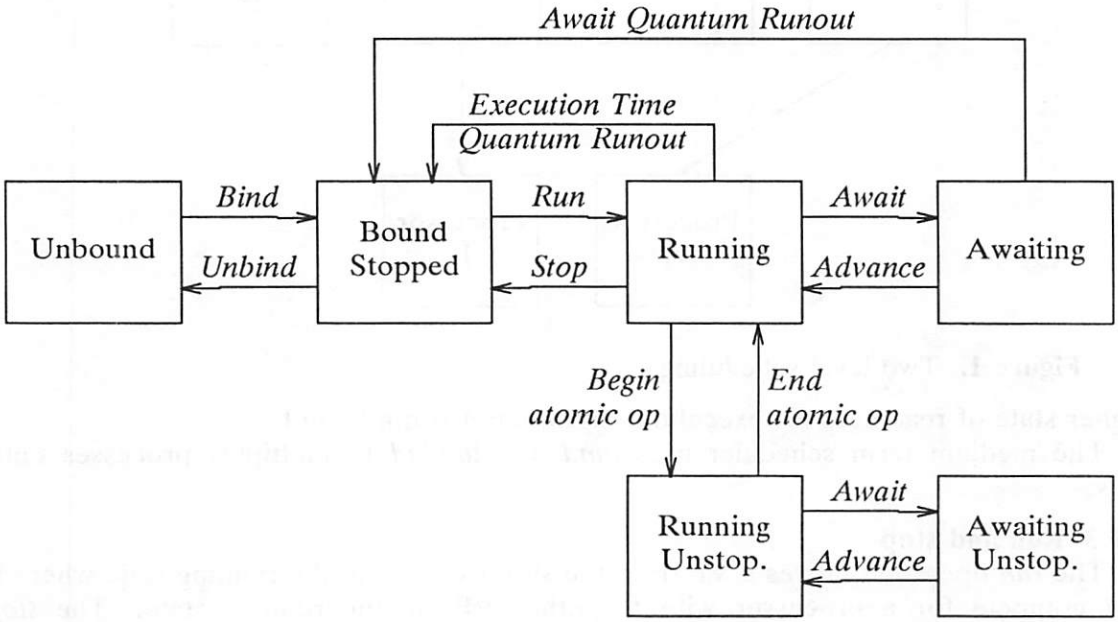


Figure 2. VP state transitions

2.2 Synchronization

The synchronization facilities used in the DG/UX kernel range from simple test-and-set to complex shared locks. Because the more complicated operations are based on the more primitive ones, and ultimately on the underlying hardware, the various synchronization abstractions are described in a bottom-up fashion. Starting with the hardware primitives, the abstractions of indivisible counter, eventcounter, and the various flavors of locks are described.

2.2.1 Hardware primitives

Two hardware primitives are available on MV/family processors: test-and-set and store conditional [Data88]. These primitives form the basis for all other synchronization mechanisms used in the system. Store conditional is often a more useful primitive than test-and-set but it is not required since it can be built using test-and-set.

Test-and-set is the classical operation that indivisibly reads the value of a bit and sets the bit to one. "Indivisibly" means that no other memory operation performed by

a processor or an I/O device can read or change the value of the bit while the test-and-set operation is in progress.

Store conditional takes three arguments: an old value, a new value, and a memory address. The operation indivisibly reads the addressed memory location, compares the memory value with the old value, and if they are equal, stores the new value into the memory location. It returns a boolean that indicates whether the new value was stored into memory. As with test-and-set, "indivisibly" means that no other memory operation can read or change the value of the location while the store conditional is in progress.

2.2.2 Indivisible counters

Indivisible counters are the next most complex synchronization primitive. They are widely used for simple counts that must be maintained in a multiprocessor system. Operations are provided to read, increment, decrement, and test the value of an indivisible counter. If two processors simultaneously initiate an increment of the same indivisible counter (such as might happen in counting the system-wide number of system calls made), the count is maintained without losing either of the increments.

Indivisible counters are implemented using the store conditional primitive. For example, the increment operation is implemented as follows:

1. The memory location is read to get the old value.
2. The old value is incremented to get the new value.
3. Store conditional is performed using the old value, the new value, and the memory address.
4. If the store conditional failed, then go back to step 1.

Other operations are implemented similarly.

2.2.3 Eventcounters

The DG/UX kernel uses a modified form of eventcounts [Reed79] as one of its basic synchronization mechanisms. An *eventcounter* is an abstraction that has a name and a value. The name of an eventcounter is simply its memory address and is unique throughout the kernel. The value of an eventcounter is a nondecrasing unsigned integer.

Closely associated with eventcounters is the concept of an *event*, composed of an eventcounter name and a value for that eventcounter. An event is said to be satisfied if the current value of the named eventcounter is greater than or equal to the value in the event. For example, an event might specify eventcounter A and value 5. If the current value of eventcounter A is 4 or less, then the event is not yet satisfied. If the current value of eventcounter A is 5 or greater, then the event has been satisfied.

2.2.3.1 Await and advance The main operations on eventcounters are *await* and *advance*. *Await* allows a process to wait for any of several events to be satisfied. If when *await* is performed, one or more of the specified events is already satisfied, the process continues execution following the call to *await*. If none of the events are satisfied, the process enters the awaiting state where it does not compete for a processor. When one or more of the awaited events are satisfied, the process will reenter the running state. If a process remains in the awaiting state continuously for more real time than its await quantum value, and its atomic op depth is zero, *await quantum runout* is said to occur and the process is stopped implicitly.

The advance operation increments by 1 the value of the specified eventcounter and awakens any processes awaiting events that are satisfied by the new value of the eventcounter. If the process associated with a satisfied event is still in the awaiting state, the VP is moved back to the runnable state. If the process associated with a satisfied event has had its await quantum expire, an "await completed" message is sent to the medium term scheduler to inform it that the process wants to be run again. Once any process has been made runnable, the dispatcher is called to see if a VP level reschedule is necessary.

The await quantum runout mechanism enables the medium term scheduler to properly manage the binding of processes to VPs. By setting the value of the await quantum, the medium term scheduler can control how long a process will be bound to a VP while it is doing nothing but waiting for an event. When the await quantum runs out the VP will be stopped, and the process can be unbound from the VP, making the VP available for use by another process. Meanwhile, when the unbound process's await is satisfied, a message will be sent to the medium term scheduler, indicating that the process is ready to run again and should be bound to a VP when convenient. This mechanism works particularly well for handling situations such as waiting for terminal input. If input is received in a few seconds, the process can continue without the overhead of being unbound; but if nothing happens after a few seconds, the process can be unbound for what may be a long wait.

Await and *advance* correspond closely to the *sleep* and *wakeup* primitives used in many UNIX kernels. The key difference is that an eventcounter "remembers" an *advance* that happens before the corresponding *await*, while if a *wakeup* happens before the corresponding *sleep*, the *wakeup* is lost and the process calling *sleep* may be pended forever. This "memory" feature greatly simplifies multiprocessor synchronization problems.

2.2.3.2 Sequencers Sequencers are an adjunct of eventcounters. The salient property of a sequencer is its value, which like an eventcounter value is a nondecreasing unsigned integer. The only operation on a sequencer is *ticket*, which indivisibly increments the sequencer value and returns the new value. The effect is like taking a number for service in an ice cream store: each process that tickets a sequencer gets a different value, thereby establishing an ordering of the processes. Sequencers are used together with eventcounters to build sequenced locks and reader/writer locks.

2.2.4 Locks

Four kinds of locks exist for general use throughout the kernel. These locks cover a range of complexity and costs for different situations. In order of increasing cost and complexity, the four types of locks are: 1) spin locks, 2) unsequenced locks, 3) sequenced locks, and 4) reader/writer locks. Each of these types is discussed in detail below.

2.2.4.1 Spin locks Spin locks are the smallest and fastest type of lock. A spin lock requires only a single bit of memory and is obtained by performing a test-and-set operation on the bit. If the bit is changed from zero to one, the lock is obtained and the process may proceed. If the original value of the bit was one, then the lock is not obtained and the process loops trying again to obtain the lock. To avoid continuously doing the test-and-set operation (which would hog the memory bus in most multiprocessor systems), the spinning is done in a separate loop that only reads the value of the bit. When the bit is observed to be zero, the test-and-set is tried again.

A spin lock is released by setting the lock bit to zero.

A process that holds a spin lock must be sure it can't be descheduled while it holds the spin lock. If the holder were to be descheduled, another process might run, try to obtain the same spin lock, and waste a significant amount of time spinning. To prevent being descheduled, a process holding a spin lock must mask interrupts, must not page fault, and must not perform an await operation. Because of these restrictions, only a few spin locks are used and only at the lowest levels of the kernel.

2.2.4.2 Unsequenced locks Unsequenced locks are the next smallest and fastest type of lock in the DG/UX kernel. An unsequenced lock requires a lock bit and an eventcounter. A contended bit is also used to speed up the case when there is no contention, which is the most common. As with a spin lock, an unsequenced lock is obtained by a test-and-set operation on the lock bit, but if the lock cannot be obtained the contended bit is set and the process awaits the eventcounter. To release an unsequenced lock, the lock bit is set to zero, and if the contended bit is set, the eventcounter is advanced. The contended bit is used to avoid advancing the eventcounter when there is no VP waiting on it.

Unsequenced locks are so named because processes waiting on the lock are not ordered. All of the processes wait on the same eventcounter value, and when the lock is released all of the processes are awakened. One of the awakened processes runs first as governed by the scheduler and obtains the lock. This scheme potentially has two problems. First, because the scheduler effectively governs which of several contending processes gets the lock when it becomes available, a low priority process may always lose out to a higher priority process and never be able to get the lock. Second, many useless reschedulings may occur as contending processes awaken, run, and await again when they discover that another process has obtained the lock first. This problem is sometimes referred to as the "thundering herd" problem, and it can be serious for a lock with high contention.

2.2.4.3 Sequenced locks Sequenced locks are similar to unsequenced locks, except that a sequencer is added to avoid the "thundering herd" problem, and the lock bit and contended bit are replaced by a counter that initially has the value 1. The lock is obtained by indivisibly decrementing and testing the counter; if the new value is zero, the lock has been obtained. If the new value is negative the process has not obtained the lock, so it tickets the sequencer and awaits the value returned by the ticket. The lock is released by indivisibly incrementing and testing the counter; if the new value is negative or zero, then one or more processes is waiting on the lock and the eventcounter is advanced.

In a sequenced lock, each waiting process waits on a different value of the eventcounter because the ticket of the sequencer returns a different value each time. Hence, when the eventcounter is advanced, only one process wakes up and it has the lock immediately by virtue of being the process that was awakened. Processes wait in line for the lock in the order in which they first tried to obtain it. However, obtaining and releasing a sequenced lock that is uncontended is slower than the corresponding case on an unsequenced lock because the indivisible operations on the counter are slower than test-and-set.

2.2.4.4 Reader/writer locks Reader/writer locks are the largest and most complex locks provided for general use. These locks may be obtained in two modes: read mode and write mode, or alternatively, shared mode and exclusive mode. The salient feature of these locks is that multiple processes may simultaneously hold a

reader/writer lock in read mode, but only one process may hold a lock in write mode. These locks are very useful for kernel data structures that are read often but seldom modified. A process obtains a reader/writer lock on the data structure in read mode to read the data structure, or in write mode to modify the data structure.

Reader/writer locks are designed to avoid starvation of a writer by many readers. If a lock is held by several readers and a writer is waiting for the lock, additional readers are not allowed to obtain the lock until the writer has obtained and released it.

The implementation of reader/writer locks is based on eventcounters and sequencers, but the complexity is such that an unsequenced lock must be used to mediate access to the internal fields. This requirement makes reader/writer locks the most expensive to obtain and release, but they are still quite useful in the situations for which they were intended.

2.3 Scheduling

Process scheduling in DG/UX occurs at three logical levels: short term scheduling, medium term scheduling, and long term scheduling. Long term scheduling is performed by the system administrator through policies on system usage based on the time of day, system load, and other factors. While it is an important part of maintaining an effective computer system, this form of scheduling is not part of the kernel proper and is not discussed further in this paper.

Short term scheduling and medium term scheduling, on the other hand, are handled by the kernel. Short term scheduling multiplexes the runnable VPs onto the available set of physical processors. The scheduling algorithms are quick and deterministic so that the overhead in finding the next runnable VP is low. Short term scheduling occurs on short intervals — 20 to 50 times per second — as processes transition between running and awaiting while doing I/O and contending for locks. The primary goal of short term scheduling is to keep the processors busy.

Medium term scheduling multiplexes user processes onto the available set of VPs. The scheduling algorithms are more complex and heuristic in order to manage the overall load on the processors, main memory, and I/O channels. The overhead in making the scheduling decisions is higher, so the scheduling occurs over longer intervals — 2 to 3 seconds — as processes enter and leave the set that is actively doing work on the system. The primary goal of medium term scheduling is to provide good response to processes that want to run while avoiding overcommitting system resources such that thrashing occurs.

The distinction between the medium term scheduling and short term scheduling can also be viewed as a policy vs. mechanism distinction. The medium term scheduler implements the scheduling policy using the short term scheduling mechanism. Different medium term schedulers could be implemented to provide different policies using the same scheduling mechanism.

2.3.1 Short term scheduling

Short term scheduling is performed by the dispatcher using the runnable VP list maintained by the other VP management code (see Figure 3). The dispatcher implements a priority based preemptive scheduling policy — the highest priority runnable VP always gets to run. The runnable list is maintained in priority order, so the dispatcher simply runs whatever VP is at the head of the list. Since the dispatcher is called whenever rescheduling might be necessary, it may find that the currently executing VP is at the highest priority and, thus, make no change.

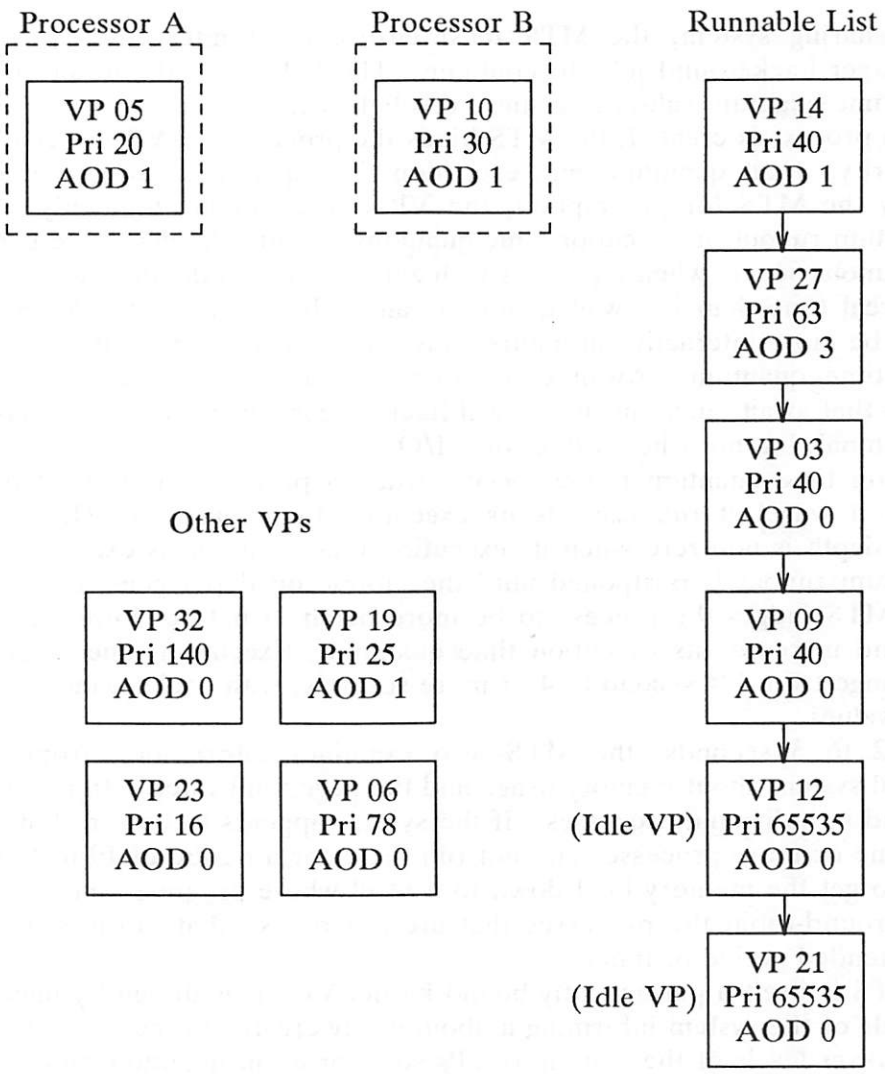


Figure 3. Short term scheduling

The runnable list contains a special idle VP for each processor in the system. These VPs are permanently bound kernel VPs that have the lowest possible priority. If there are no higher priority VPs that have useful work to do, one of the idle VPs is found and run.

A key point about the dispatcher is that VPs may be involuntarily descheduled while executing in the kernel. This feature is necessary because many important parts of the kernel execute as special kernel VPs that must receive timely service for the system to perform reasonably. As described later, I/O interrupts are effectively mapped to VP scheduling operations, so good response is necessary.

2.3.2 Medium term scheduling

The medium term scheduler (MTS) assigns processes to VPs based on various scheduling parameters and heuristics. Fundamentally, the MTS assigns processes to VPs starting with the highest priority process and so on until no more VPs are available. But the MTS also collects information about processes as they run and adjusts the processes's priorities to achieve favorable system operation. As DG/UX

is a time-sharing system, the MTS must ensure good interactive response while allowing larger background jobs to continue. The MTS uses the await quantum and execution time quantum values to achieve this balance.

When a process is created, the MTS binds the process to a VP and runs it with an initial priority, await quantum, and execution time quantum. If it is not explicitly stopped by the MTS for preemption, the VP will eventually implicitly stop due to await quantum runout or execution time quantum runout. As described eariler, await quantum runout occurs when a process with zero atomic op depth awaits continuously for more real time than its await quantum value. In this case, the MTS judges the process to be more interactive in nature, raises the process's priority, and reduces its execution time quantum. Await quantum values are typically a small number of seconds so that await quantum runout will likely occur when a process waits for input from a terminal, but not when it does disk I/O.

Execution time quantum runout occurs when a process's accumulated execution time since it was last *run* exceeds its execution time quantum. (If the process's atomic op depth is non-zero when its execution time quantum is exceeded, execution time quantum runout is postponed until the atomic op depth goes to zero.) In this case, the MTS judges the process to be more batch in nature, lowers the process's priority, and increases its execution time quantum. Execution time quantum values typically range from 1/4 second to 4 or more seconds, with background jobs receiving the larger values.

Every 2 to 3 seconds, the MTS also examines information from the virtual memory subsystem about memory usage and the page fault rate, both in the system as a whole and in individual processes. If the system appears to be thrashing, the MTS will pick one or more processes and not run them for a period of 10 to 15 seconds in an effort to get the memory load down to a level where progress can be made. The MTS will round-robin the processes that are not run so that fairness is maintained over an extended period of time.

The MTS is itself a permanently bound kernel VP. It is driven by messages from higher levels of the system informing it about newly created processes to be scheduled and from lower levels of the system as VPs stop for await quantum runout, execution time quantum runout, or the completion of an await. It is also timer driven to wake up every 2 to 3 seconds even if there are no messages so that system memory usage can be examined and the scheduling mix adjusted.

The details of the heuristics used by the MTS and the experimental results with various values of the scheduling parameters are beyond the scope of this paper. A future paper is planned to cover these topics.

3. Input/Output

The design of device level input/output in the DG/UX kernel is significantly influenced by the potential for multiple processors to access devices and take interrupts simultaneously. Interrupt handling in particular is different from what is found in many UNIX operating systems, and one or more asynchronous I/O daemons have been introduced.

3.1 Device/processor interaction

3.1.1 Hardware model

The DG/UX hardware model of device I/O is fundamentally symmetric in the way that the operating system initiates an I/O operation to a device and in the way that

device responds when the operation is complete. No part of a device driver ever knows or cares on which processor it is running.

First, all processors can access all devices in the system. This capability is important in avoiding a bottleneck in the system that will occur if an important device can only be accessed by a particular processor. Much strange and error prone code can be avoided if a process can simply call into the device driver and initiate an operation on a device as if it were in a uniprocessor system. The device driver must contain the appropriate locks to prevent two processors from trying to initiate an operation on the same device at the same time, but two processors can and should be able to execute the same driver code simultaneously if they are manipulating two independent devices of the same type.

Second, when a device completes an operation, the hardware may chose to interrupt any of the processors. The processor chosen need not be the same processor that initiated the operation on the device, and two different interrupts from a device may be directed to different processors at different times.

Third, each processor may mask all interrupts directed to it. If an interrupt is directed to a processor that has interrupts masked, the interrupt is pended and will be taken when the processor unmasks interrupts again. Multiple interrupts may be pending to a processor without any interrupts being lost. Note that when a processor masks interrupts, other processors' interrupt handling is not affected.

Finally, each device in the system may be masked to prevent the device's interrupts from being recognized on any processor in the system. If a masked device completes an operation, its interrupt is pended until it is unmasked, whereupon the normal interrupt assignment mechanism will direct the interrupt to a particular processor. Any processor may mask and unmask any device, and the masking and unmasking need not be done by the same processor.

3.1.2 Interrupt handling

The main goal of all interrupt handlers is to map the interrupt into an advance of the appropriate eventcounter. The interrupt handler may read device status registers and store the results, but it does not try to interpret the results except to determine which eventcounter to advance. A process waiting on the eventcounter will be awakened, process the results stored by the interrupt handler, and start the next request on the device. Since no real work is done in the interrupt handler, synchronization problems between interrupt handlers and the rest of the kernel are almost non-existent. (The implementation of the eventcounter *advance* operation is the main exception.) The interrupt handler synchronization problems have been mapped into problems of concurrent access by multiple processes, which must be solved anyway in a multiprocessor system.

When an interrupt occurs on a processor, the state of the currently running process (i.e., virtual processor) is pushed onto the process's kernel stack and the interrupt handler borrows that same stack for its execution. The system has no separate interrupt stacks. Because the interrupt handler will do little or nothing except advance an eventcounter, there is no need to nest interrupts in order to meet latency requirements. Hence, the amount of extra space required on every process's kernel stack for handling interrupts can be bounded rather tightly. Furthermore, if the advance of an eventcounter by an interrupt handler should cause a higher priority process to become eligible to run, the current process can be simply descheduled with the interrupt handler stack frame still on its stack as if it had called the interrupt handler as a subroutine. When the process runs again, the interrupt frame will be

popped off the stack and it will pick up execution where it left off previously.

3.1.3 Synchronization

If a device may have multiple I/O requests outstanding, as on a disk controller with multiple units that can be seeking simultaneously, contention for controller registers may arise between a process that wants to submit a new request to the controller and the interrupt handler that is servicing the completion of a different request. The contention may occur with the new request being interrupted by an interrupt handler running on the same processor, or the new request and the interrupt handler may be proceeding in parallel on different processors.

A standard locking paradigm solves this problem. The device driver has a lock associated with the controller and this lock must be held whenever the controller registers are to be accessed. Even if contention with the interrupt handler did not occur, this lock would likely be necessary to mediate access to the device by two processes trying to submit requests. The following pseudo-code fragments show the use of the lock by the standard parts of the device driver and by the interrupt handler portion:

Non interrupt handler	*Interrupt handler*
mask_device();	if (!obtain_lock_no_wait(&controller_lock))
obtain_lock(&controller_lock);	return;
...	
(manipulate controller registers)	...
...	(manipulate controller registers)
release_lock(&controller_lock);	...
unmask_device();	release_lock(&controller_lock);

Before non-interrupt handler code obtains the controller lock, interrupts from the device must be masked so the device is prevented from interrupting any processor. (Masking interrupts on the executing processor instead doesn't work because the device could interrupt a different processor.) Even with the masking, though, the lock may be held when the interrupt handler is entered. This situation can occur if the interrupt occurs on one processor and it starts processing the interrupt, but another processor (not running in the interrupt handler) gets the lock before the interrupted processor can. In this case the interrupted processor simply returns because the interrupt will occur again when the other processor releases the lock and unmasks the controller.

This interrupt handling scheme offers several advantages in a multiprocessor system. First, interrupt handlers are simple and easy to write; access to controller registers can be mediated by a standard locking paradigm if necessary. Second, because interrupts are mapped into eventcounter advances which result in process reschedulings, the rest of the kernel doesn't have to worry about interrupts as a separate problem. Third, since much code that is traditionally executed in an interrupt handler is here executed as part of a scheduled process, interrupt latency and priority can be controlled through the process scheduling mechanism instead of a nested interrupt scheme that becomes messy in a multiprocessor. Finally, the interrupt handling is truly symmetric and scales as the number of processors and devices is increased. No bottleneck occurs as it would if all interrupt traffic were

directed through one processor.

3.2 Asynchronous I/O daemon

In order to handle asynchronous I/O requests (i.e., requests in which the calling process does not wait in the driver for the request to be completed), an I/O daemon process must be introduced to act on behalf of the original requestor. The I/O daemon is a permanently bound kernel process with a higher priority than any user process. In a multiprocessor system, multiple I/O daemons may be appropriate to avoid single threading the handling of asynchronous I/O requests.

The daemon continuously takes messages off a global queue and performs the request in the message. A request takes the form of a function to call and an argument to pass to that function. When an interrupt comes in from a device operation that was started by an asynchronous request, the interrupt handler queues a message to the daemon and advances the daemon's eventcounter instead of the eventcounter associated with a process waiting in the driver. The message specifies a function in the device driver to process the completion of the operation. The daemon performs the cleanup of the operation that would be performed by the requestor in a synchronous request. If the original request is now complete, the device driver routine will typically call a higher level routine that will advance an eventcounter to notify the higher level that the original asynchronous request is now complete.

4. Virtual memory

Most virtual memory issues in a multiprocessor kernel are tied to the memory management hardware. The hardware model used by DG/UX is described followed by the particular issues of maintaining address translation cache consistency. The purging daemon is then presented as another example of a kernel virtual processor.

4.1 Hardware model

Virtual memory design is of necessity directly tied to a particular hardware model because of its close association with the hardware. In DG/UX, two assumptions are particularly important. First, the kernel assumes that the memory management hardware maintains consistent views of physical memory on all processors and across I/O operations. Thus, data and instructions caches are presumed to be invisible to the operating system. Second, each processor in the system is presumed to have its own address translation cache for mapping logical addresses to physical addresses, and these caches are not kept consistent by the hardware as the operating system moves pages in and out of memory. The proper management of the address translations caches is particularly important to the kernel.

4.2 Address translation cache

The difficulty in address translation cache management on a multiprocessor arises because a processor may remove a page from memory for which a translation exists in one or more of the other processor's caches. Furthermore, it is generally not possible to query the other processors' caches to determine if an entry needs to be removed. The brute force approach to solving the problem is to cross-interrupt all other processors to request that they flush their caches every time a page is removed from memory. This approach is expensive, particularly if a selective cache flush is not possible, and it gets worse as the number of processors grows.

The kernel reduces cache flush traffic considerably by keeping a record of the entries that could possibly be in a particular processor's cache. Each time the dispatcher schedules a new process on a processor, it flushes the processor's cache and records in a global array the identity of the process running on that processor. When a page is to be removed from memory, the page is first identified as belonging to a particular process or to the kernel, and hence to all processes. If the page belongs to a particular process, the global array identifying the processes running on each processor is then searched, and if a match is found, only that processor is interrupted to perform a cache flush. In most cases, no match is found and thus no cache flush is needed. On the other hand, if the page being removed belongs to the kernel, all processors must be flushed and the brute force method is used.

Since some hardware will support the purging of single cache entries without disturbing the others, processor-to-processor mailboxs are maintained so that when a processor is requested to flush its cache, it can get the details of the page being removed from memory and perform a selective cache flush.

4.3 Purging daemon

Removing pages from memory is the responsibility of the purging daemon, a permanently bound kernel VP. The purging daemon is awakened by an eventcounter advance when a process uses a free frame and the number of free frames in the system is below a certain level. The purger finds pages to remove from memory and performs the appropriate cache flushes until the number of free pages is back to an appropriate level. In a multiprocessor system with 4 or more processors, multiple purging daemons may be appropriate to avoid single threading page removal operations.

5. Conclusions

The DG/UX 4.00 kernel is in production use at customer sites on a range of MV/family processors, including dual processor models. It has achieved its goals of being a fully symmetric UNIX kernel that makes effective use of multiprocessor hardware. The virtual processor model provides an excellent basis for the rest of the kernel in synchronization and in scheduling. The multiprocessor I/O model also moves well from uniprocessor to multiprocessor systems and simplifies the interactions between interrupt handlers and base level code.

6. References

[Arno74] Arnold, J. S., D. P. Casey, and R. H. McKinstry. "Design of tightly-coupled multiprocessing programming" *IBM Systems Journal*. Vol. 13 No. 1 (1974), 60-87.

[Bach84] Bach, M. J. and S. J. Buroff. "Multiprocessor Unix Operating Systems" *AT&T Bell Laboratories Technical Journal*. Vol. 63 No. 8 (October 1984), 1733-1749.

[Data88] Data General Corporation. *ECLIPSE MV/Family (32-Bit) Systems Instruction Dictionary*. Ordering No. 014-001372. Revision 00, January 1988.

[Ensl77] Enslow, Philip H., Jr. "Multiprocessor Organization — A Survey" *ACM Computing Surveys*. Vol. 9 No. 1 (March 1977), 103-129.

[Gobl82] Goble, George H. and Michael H. Marsh. "A Dual Processor VAX 11/780" *Conference Proceedings, The 9th Annual Symposium on Computer Architecture*. (1982) 291-298.

[Hami88] Hamilton, Graham and Daniel S. Conde. "An Experimental Symmetric Multiprocessor Ultrix Kernel" *Usenix Conference Proceedings, Winter 1988*. The USENIX Association, Berkeley, CA., 1988, 283-290.

[Holl79] Holley, L. H., R. P. Parmelee, C. A. Salisbury, and D. N. Saul. "VM/370 asymmetric multiprocessing" *IBM Systems Journal*. Vol. 18 No. 1 (1979), 47-70.

[Nogu75] Noguchi, Kenichiro, Isao Ohnishi, and Hiroshi Morita. "Design considerations for a heterogeneous tightly-coupled multiprocessor system" *AFIPS Conference Proceedings, 1975 National Computer Conference*. 561-565.

[Reed76] Reed, David P. "Processor Multiplexing in a Layered Operating System" MIT/LCS/TR-164, Laboratory for Computer Science, Massachusetts Institute of Technology, June 1976.

[Reed79] Reed, David P. and Rajendra K. Kanodia. "Synchronization with Eventcounts and Sequencers" *Communications of the ACM*. Vol. 22 No. 2 (February 1979), 115-123.

MOSIX: An Integrated Multiprocessor UNIX

Amnon Barak and Richard Wheeler

Department of Computer Science
The Hebrew University of Jerusalem
Jerusalem 91904, Israel

barak@humus.huji.ac.il

ABSTRACT

MOSIX is a general-purpose Multicomputer Operating System which Integrates a cluster of loosely connected, independent computers (nodes) into a single-machine UNIX* environment. The main properties of MOSIX are its high degree of integration and the possibility of scaling the configuration to a large number of nodes. Developed originally for a network of uniprocessor nodes, it has recently been enhanced to support nodes with multiple processors. In this paper we present the hardware architecture of this multiprocessor workstation and the software architecture of the MOSIX kernel. We then describe the main enhancements made in the multiple-processor version and give some performance measurements of the internal mechanisms of the system.

1. INTRODUCTION

This paper describes a symmetrical multicomputer operating system, called MOSIX, which integrates a cluster of independent computers into a single-machine UNIX environment. By restructuring the UNIX kernel into machine-dependent and machine-independent parts, each MOSIX kernel isolates the user's processes from the specific processor in which they execute, while at the same time providing these processes with the standard UNIX interface [1,2]. In order to provide efficient network-wide services, different MOSIX kernels interact at the system-call level. This means that processes are executed in a site-independent mode and that all system calls are executed in a uniform way, regardless of the initiating site and the site which has the target object. The organization of the MOSIX kernel allows a unified, network-wide interaction among heterogeneous computers, i.e., it can be implemented on any hardware. Another outcome of this organization is the possibility for process migration among (homogeneous subsets of) processors to improve the response time.

The other main property of MOSIX is the possibility of scaling the configuration to a large number of processors. This is achieved by ensuring that all kernel interactions involve at most two processors, and by limiting the network-related activities at each node, using probabilistic algorithms. In other words, the design of the internal management and control algorithms in MOSIX is such that each processor has a fixed amount of overhead, regardless of the size of the network.

The hardware configuration for MOSIX is a cluster of loosely-coupled, independent computers (nodes), which may consist of any combination of uniprocessor nodes, shared-memory multiprocessors, and shared-device multicomputers, interconnected by a communication link. This communication link is used for kernel-to-kernel interaction. In order to provide high

* UNIX is a trademark of AT&T.

reliability and efficiency and to support scaling, MOSIX uses private communication protocols which have been carefully designed to guarantee bus performance and reliability over a local area network (LAN).

The original version of MOSIX , called MOS, was compatible with UNIX Version 7 [3]. It was developed in 1982 for a cluster of PDP-11 computers interconnected by ProNET-10, a 10 MBit/sec LAN. This system was ported in 1984 to a cluster of CADMUS/PCS, MC68000-based computers. Later versions of MOSIX were redeveloped for Digital's VAX and National's 32000 families of computers connected by Ethernet. These versions are compatible with AT&T UNIX Version 5.2. In all of these configurations, each node is a uniprocessor, independent UNIX system, with complete hardware and software facilities, while the whole cluster behaves like a single UNIX machine.

Recently, MOSIX was enhanced to handle nodes with several processors. As in the uniprocessor case, MOSIX can be used to integrate a cluster of several such nodes. However, in this recent implementation, each node is a multiprocessor system with several independent processing elements, each with its own local memory. This paper deals with the architectural and software enhancements of this multiprocessor system. Since in our configuration several processors are placed in the same enclosure, we call this multiprocessor node a "workstation". The term "processor" refers to a single processing element within a workstation.

The main characteristics of the workstation MOSIX are:

- **Replicated kernel**: the system kernel is replicated in each processor
- **Transparency**: the bus and the network are completely invisible at the user level
- **Decentralized control**: each workstation makes all its control decisions independently
- **Autonomy**: each workstation is capable of operating as an independent system
- **A unified file system**: the file system consists of one tree with a replicated 'super-root'
- **Scaling**: probabilistic algorithms are used for system management and network related overhead at each node is limited
- **Adaptive load balancing**: the system initiates process migration within the workstation and among different workstations to improve performance
- **Dynamic configuration**: workstations may be added or removed at any time and processors may fail with minimal effect on the system
- **Compatibility**: the system is compatible with AT&T UNIX Version 5.2

Some details about these properties are given in Section 3. More details can be found in our earlier papers and reports [3,4,5,6].

The motivation behind the MOSIX project has been research and development in distributed operating systems, including the exploration of kernel architectures, internal-system mechanisms, communication protocols, control algorithms, scaling and the support of concurrency, and multiprocessing. The resulting system is intended for large multiuser environments, for users executing a large number of independent tasks, e.g., multiple windows, and for applications that can benefit from concurrent processing with coarse or medium granularity using a large number of nodes [5]. Other projects that address the scaling issues are Grapevine [13], a distributed registry system which serves as a repository of naming information for an electronic mail system, and Andrew [10], a large distributed environment built at Carnegie Mellon University. Other projects with similar goals to MOSIX are Locus [12] developed at UCLA, the V kernel [7] developed at Stanford University, and Sprite [11], developed at UC Berkeley.

This paper is organized as follows: in Section 2 we describe the architecture of the workstation and the specific enhancements made at the board level to support multiprocessing. Due to the internal architecture of the MOSIX kernel, relatively few changes were necessary in order to generalize the uniprocessor version into the workstation system. In Section 3 we describe this architecture and present the mechanism for performing remote operations between different kernels. Some of the important characteristics of MOSIX are also described. In Section 4 we describe the enhancements made to the workstation version of MOSIX, and in Section 5, we give performance measurements of system calls, data transfers and process migration among the processors of a workstation and between workstations. Our conclusions are given in Section 6.

2. THE HARDWARE CONFIGURATION

The hardware configuration for MOSIX consists of a cluster of workstations connected by the ProNET-80, an 80 Mbit/sec token-ring-based LAN. This network is used for internal, kernel-level communication among the workstations, while an Ethernet LAN connected to at least one workstation in the cluster is used for external, TCP/IP communication. The workstation consists of up to eight processors, each with its own local memory and a floating-point unit. All the processors of a workstation share I/O devices and communication controllers over a VME bus, which can also support an optional shared memory. The rationale behind this architecture is an improved cost/performance ratio over the single-processor workstation.

The specific processor used is National's NS32532 microprocessor. The current measurements were carried out on a configuration with two 30MHz (10 MIPS) processors and several 25MHz (8 MIPS) processors. Each processor has 4 MBytes of local memory, a 64-KByte on-board cache, 2 serial lines, a PROM, and an NS32381 floating-point unit. A Weitek WTL3164 floating-point chip and an additional 12 MBytes local memory board are optional. The board level, called the VME532, is a 2-board set incorporating a full VME bus interface, including a master-slave (A32/D32), a location monitor, and a 4-level system controller. In addition, all read/write operations to I/O devices are non-cacheable to both on-chip and external caches. This ensures that all data transfers from/to I/O devices will directly reach the device.

The VME532 supports multiprocessing in configurations that include up to 16 processors in one VME card cage. The specific enhancements that were made at the board level include:

1. A unique ID for each processor in the range 0-F. The ID is set by a thumb-wheel switch that can be read by the software.
2. Each processor has its own cacheable, dual-ported local memory. This memory is mapped into the VME address slot determined by the processor ID. The local memory of each processor can be accessed by other VME masters (processors) in a non-cacheable mode.
3. A location monitor that interrupts the CPU for efficient inter-processor interrupts.
4. A VME bus watcher that maintains consistency between the main memory and the caches.
5. 240-252 MBytes of shared VME address space, cacheable by all the processors.

The multiprocessing support implies that the bus activities, including the I/O interrupts, are available to all the processors. It is the duty of the software level to coordinate the management and synchronization of these activities. One way to organize these activities is by using a software-level master/slave organization. The outcome of such an organization is improved reliability, since any processor can assume the role of the master. If this master crashes, then another processor can take on its role. Note that in this case the workstation must be re-booted. Also note that if a slave processor crashes, the system is not damaged, but processes running at this slave are lost. More details about the software organization are given in Section 4.

3. THE ARCHITECTURE OF THE KERNEL

MOSIX is a symmetrical multiprocessor operating system obtained by restructuring the UNIX kernel while preserving the standard UNIX kernel interface. Each processor running MOSIX is an independent UNIX machine. It has a complete copy of the kernel, with the possible exception of device drivers. In this section we describe the architecture of the MOSIX kernel for a multiprocessor consisting of many uniprocessor independent computers.

The MOSIX kernel consists of three parts: the machine-dependent part, called the *lower-kernel*, the machine-independent part, called the *upper-kernel* and the communication layer, called the *linker* [3]. The *lower-kernel* contains routines that access local resources such as device drivers for local disks, and routines which access file and process structures. The *lower-kernel* is tightly coupled with the local processor, it has complete knowledge of all the local objects, and it can access only these local objects (or objects that have migrated to that processor). The *lower-kernel* does not have any knowledge about processors other than the one in which it executes, and it does not distinguish between requests originating in its local processor or requests originating in other processors. The *upper-kernel* executes in a machine independent mode. On one hand, it provides the standard UNIX system call interface. At the same time, the *upper-kernel* does not know the processor ID in which it executes, but it has a complete knowledge about the locations of all the objects it handles. For example, when a process executes a system call, the *upper-kernel* performs the preliminary processing of the parameters, e.g. calling *namei* to parse a path name into a *universal inode*, or checking that the user has permission to access a certain object. Eventually, the *upper-kernel* calls the relevant remote kernel procedure using the remote procedure call (RPC) mechanism to complete the service. For example, in the following the remote kernel procedure "proc_name" is executed with the parameters specified in "param_list":

Rproc_name (machine_id, param_list).

Next, the call is passed by the *upper-kernel* to the *linker* which decomposes the RPC into the procedure name and the parameter list. The *linker* examines the first parameter of the call to determine where the call needs to be executed. If it is a remote call, the *linker* encapsulates the call into a message and sends the message over the network. If the call can be executed locally, the *linker* invokes the local *lower-kernel* procedure directly. On the target machine, an *ambassador process*, a lightweight kernel process, executes the appropriate *lower-kernel* procedure for the calling process. The result of the system call is then encapsulated into another message and returned to the calling node.

The communication protocol used by the linkers is designed to provide a high degree of reliability. This results from the fact that in MOSIX the information passed between the linkers is critical. For example, a change of one bit could cause the removal of a file rather then closing it. The specific protocol that is used includes a CRC checksum which is validated by the receiver, a software-initiated acknowledgement message in addition to the hardware acknowledgement provided by the token-ring network, a message ordering for each pair of nodes, and a password for each message.

In order to support dynamic reconfiguration, e.g. removing or adding new machines, each remote access generates a new remote kernel call which returns a special error code if the node is currently unreachable. The system does not require any special action to incorporate a new node: the first remote call to the newly joined node is simply sent by the system in the usual way, and the connection is created dynamically.

Remote system calls which need to transmit large amounts of data in addition to the result of the RPC use the *funnel* mechanism to copy data from the memory space of one machine to another. The *linker* handles the implementation of funnels by breaking up large blocks of data

into message-size pieces at one end, sending the messages over the network, and reunifying the data in the proper order on the receiving machine. The data is then copied by the *linker* of the receiving node directly to the specified user's address space. For example, for the *read* system call, the *upper-kernel* sets up an input funnel on the local machine before calling the remote *Rread* system call. If the system call accesses a remote file, the *linker* routes the call to the target machine, and an ambassador process there calls the appropriate kernel procedure for reading a file. As each logical file block is read, the data is placed into the remote end of the funnel and passed back to the initiating machine's *linker*. After the call is completed, the returned status of the system call is encapsulated by the remote *linker* and passed back to the calling machine.

The net result of this kernel architecture is that the user's process is completely isolated from the processor in which it is currently running. This isolation requires all kernel references to be made in a machine-independent (universal) mode, a unified naming scheme, and good performance from the RPC mechanism. Note that this organization can be applied to any version of UNIX and can easily support heterogeneous hardware. The only parts which has to be implemented for each machine type is the *lower-kernel*. Another outcome of this organization is a simple process-migration mechanism.

The main characteristics of the uniprocessor MOSIX include network transparency, decentralized control, dynamic configuration, a distributed file system, and load balancing by dynamic process migration. Details about these characteristics can be found in [3]. For completeness, we give below a short description of the file system [4], the load-balancing mechanism [6], and some of the probabilistic algorithms.

3.1. The file system

As in UNIX, the file system of MOSIX is a forest, with several disjoint trees. Each tree is a complete UNIX file system, with regular files and devices as leaves, and directory files as internal nodes. Each user is assigned to one of these trees, and the root of that tree is the root directory of this user. In this scheme, a user always has the same root (home) directory at all login sessions, but different users may have different root directories [4].

The MOSIX file system uses a super-root, "**/...**", as a network-wide root. The super-root is replicated in all the nodes, and it is supported by the kernel. When addressing a file using an absolute path, mainly on a tree which differs from the user's root directory, the user prefixes the super-root to the machine name and then adds the usual UNIX path. For example, "**/.../m2/etc/passwd**" is the absolute path name for the password file on machine number 2. One version of MOSIX uses a special file type which stores the remote-machine number in its inode. When such an inode is accessed by the kernel, the inode of the special file is automatically replaced with the inode of the root directory of the remote node indicated. For example, if the special file "**/usr/systems/bert**" is created as a remote escape to the file system on machine m2, the path "**/usr/systems/bert/etc/passwd**" refers to the password file on machine m2. Note that this change eliminates the need for a special, non-standard UNIX pathname syntax. Also, it allows the MOSIX file system to have arbitrarily placed links between nodes.

In MOSIX, the *inode* of an open file is held by the site at which the file resides. All remote *opens* are returned with a universal pointer to the file. This *universal file pointer* includes the identity of the machine in which the file resides, and it is used for future file accesses. We note that MOSIX does not support file migration. The garbage-collection algorithm is used to clean up allocated inode structures in case a failure occurs in the calling node.

3.2. Load balancing

In MOSIX, load-balancing is carried out by dynamic process migration [6]. As a result of the system architecture, a process running under MOSIX is not sensitive to its physical location:

system calls which access resources not located on its current node are automatically forwarded by the *linker* to the remote node. In order to support process migration, the hardware configuration must consist of clusters of homogeneous processors. Process migration is allowed only among processors with the same instruction set, because application tasks may be assigned or migrate arbitrarily among the nodes of the cluster. Note that all of the other functions of MOSIX may be supported among heterogeneous processors.

In MOSIX, each processor exchanges its own local load estimate with that of a set of randomly selected processors every unit of time (one second in the current implementations). Load estimates received from other processors are kept in a *load vector* and are "aged" to reflect their decreasing relevancy. The algorithms used for load balancing are probabilistic, and are intended to provide each node with the latest, up-to-date information about the loads of other nodes. As proved in [9] this is achieved in $O(\log N)$ units of time for an N-processor system. Another goal of these algorithms is to overcome a node failure and to be responsive to dynamic configuration.

The decision to migrate a process is based on many parameters, including the past profile of the process, the amount of local-versus-remote I/O, the relevant locations of this I/O, the relative load of the sending and receiving nodes, the size of the process, etc. After a decision has been made to migrate a process, the target processor may still refuse to accept the migrating process if it so desires (due to local circumstances). Processes which have a history of "forking" new child processes are given migration preference by the algorithm to further speed up the even distribution of the load. Also, processes with a history of I/O operations to some specific set of nodes are given priority for migration to one of these nodes.

Several heuristics are used to avoid over-migration. To begin with, the load-balancing algorithm does not migrate a process unless it has accumulated a minimal amount of CPU time on the current processor. Also, the load estimate sent out to other processors, or *export load*, is slightly higher than the true local load. Each process that has been allowed to migrate to a processor is immediately counted in the receiving processor's *export load* estimate and removed from the local load estimate of the sending processor. New requests for migration are accepted only if the difference between the revised load estimates is still greater than a known threshold value. The actual migration starts by passing the necessary information required to re-build the page table, followed by the data pages which were changed, i.e., the "dirty pages", including in-core or swapped-out pages. The rest is either mapped to the original file, or defined as "all zeroes". Note that after the process migration is completed, no "tail" remains in the sending site.

3.3. Distributed probabilistic algorithms

In order to limit the network-related management activities performed by each node, the internal control algorithms of MOSIX are distributed. These probabilistic algorithms attempt to achieve near optimal performance at the fraction of the cost required by a centralized algorithm. For example, the load-balancing algorithm described above ensures that each processor has sufficient, but limited amount of information about other processors. This method allows load-balancing interactions between many different pairs of processors while at the same time it eliminates flooding of a single processor, for example, immediately after it joins the network.

Another example of a distributed algorithm is the remnant-collection mechanism. Unlike the more traditional garbage-collection mechanism, the MOSIX algorithm is intended to remove any remnants of a process that lost its objects or communication links due to a failure in a remote node. For this purpose, all the resources allocated in MOSIX include a timer and a *"keep-alive"* mechanism which is supported by the kernel. The timer is reset at regular intervals and upon each interaction between the process and the object. If the timer expires, then the object is removed, since the process that uses this object does not exist any longer.

4. THE WORKSTATION VERSION

The workstation version of MOSIX, like all previous versions, is a symmetric system, in the sense that each workstation is an independent computer with full UNIX capability. This symmetry can be extended to the processor level if sufficient I/O and communication controllers are available. In order to simplify the organization of the software level and to use shared I/O and communication devices, a master/slave organization of the workstation software was chosen. This implies a point of asymmetry since one processor (processor 0) assumes the role of a **master** while the remaining processors are **slaves**, i.e., they need the master processor to coordinate some of their I/O and communication requests. We note that in the implementation of the workstation kernel a great effort was made to minimize this dependency. For example, slave-to-slave (within the same workstation) communication and load balancing is done without the interference of the master processor. Also, in swapping, data is directly transferred between the slave processor and the disk, thus reducing the amount of overhead required from the master processor.

The main enhancements of the workstation kernel were made in the communication interface at the *linker* level and in the boot procedure. Recall that the local memory of each processor can be addressed directly by all the other processors (in the same workstation). For example, when the workstation is rebooted, the master processor loads the kernel from the disk, then copies the kernel to each slave processor using the direct-addressing mode. Another mechanism that was modified in the workstation version is the handling of the file-system buffer cache. In the current version, the master processor handles all of the file system operations since the slave processors do not have a file system of their own. Similarly, a new process ID (PID) is assigned only by the master processor.

Only minor modifications were necessary to enhance the single-machine kernel to the multiprocessor version. In fact, due to the architecture of the kernel, most of the kernel routines were used without any change. The only modifications that were made resulted from the hardware architecture of the workstation. More specifically, changes were made to the I/O and communication routines, and to the boot function described earlier, and to the system administration. The other main changes that were made were:

1. Remote swapping allows slave processors to swap directly to the disk in the same workstation. In this case the master processor serves only as coordinator, but the actual addressing and block allocation is done by the slave. For this purpose each slave processor has an allocated swap space on the disk.
2. A multi-level load-balancing scheme between master and slave processors. The algorithms of the load-balancing mechanisms were tuned to include master-to-master, master-to-local-slave, master-to-remote-slave, and slave-to-slave migration (in the same workstation and in different workstations). Each case requires a different set of migration parameters. These parameters were provided by the automatic tuning mechanisms developed for this purpose.
3. A two-level *linker* for master and slave-oriented commands. The linker was enhanced to include all possible combinations of data transfers and commands between local and remote master and slave processors.

A TCP/IP package for external communication, including remote terminal access, file transfer, and remote command execution was also added. The network level supports the Internet Protocol (IP) and the Internet Control Message Protocol (ICMP). The host level supports the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). Applications supported include the TELNET protocol for connecting remote terminals (RFC), and the File Transfer Protocol (FTP). Also supported are the UNIX special services such as *rcp* (remote file copy), *rlogin* (remote login), and *rsh* (remote execution of commands). The application programmer has access to the host level protocol using a BSD UNIX 4.2-like socket software library.

5. PERFORMANCE MEASUREMENTS

In this section we start by comparing the time required to perform local vs. remote system calls, both between different processors of the same workstation and between different workstations. The system calls were measured by a set of benchmarking programs which execute each particular call on a remote object and a local object 100,000 times. We also present measurements of the rate of data transfers and process migration. The section concludes with a performance analysis of a distributed implementation of the traveling-salesman problem.

First, we note that due to the architecture of the MOSIX kernel, many system calls are performed locally by each processor without any slowdown penalty. Examples of system calls of this type are *getpid*, *signal* and *time*. File-related system calls are always performed by a master processor because of the architecture of the workstation. File-related system calls which access a file on some remote workstation are handled by the remote workstation's master. The times required to perform the most frequently used file-related system calls are given in Table 1. All times are given in milliseconds. The first column shows the time required by a process running on the master processor to access a local file. The second column shows the time required by a process running on a slave processor in the same workstation to access the same file, i.e., via the VME bus. The third column shows the time required by a process running on one master processor to access a file located in another workstation, i.e., master-to-master, using the ProNET-80. The fourth column shows the slowdown factor associated with the VME bus (master-slave in the same workstation). It is computed as the slave time (column 2) divided by the master time (column 1). The fifth column is the remote-master time (column 3), divided by the slave time (column 2). In both cases, the involved processors ran at 30MHz.

System call	Local call Master	Slave to local Master	Master to remote Master	VME Slowdown Ratio	ProNET Slowdown Ratio
read (1KByte)	0.34	1.36	1.93	4.00	1.42
write (1KByte)	0.68	1.65	2.33	2.43	1.41
open & close	2.06	4.31	5.19	2.09	1.20
fork (256KByte)	7.80	21.60	23.10	2.77	1.07
exec (256KByte)	25.30	51.50	53.80	2.04	1.04

Table 1: Local vs. remote system call execution times (ms)

One useful measure of the overall effect of remote access on a process is a *weighted average* of the slowdown factors. Using the frequencies of system calls measured in [8], the frequency of each call is multiplied by the slowdown factor associated with remote execution. The results show a weighted slowdown factor of 2.8 for system calls executed by the slave versus the same set of system calls executed by the master. Comparison of the time required to perform system calls between a master and slave in the same workstation, using the VME bus, and the time required between two masters in different workstations, using the ProNET LAN, shows a slowdown of only 32%. Recall that the LAN protocol uses a software-generated CRC at both the sending and receiving nodes, as well as an acknowledgement message. Thus it is estimated that in a network that supports this protocol at the hardware level there would be no slowdown factor, other then the one generated by the VME bus. This indicates that the configuration can be scaled up with only a slight degradation in performance.

In Table 2 we give the speeds of several data-transfer mechanisms of MOSIX. First, we give the rate of a memory-to-memory copy using the *funnel* mechanism between processes

running on the same processor. The next figure shows the throughput of a memory copy between a process that runs on a master processor and a process that runs on one of its slave processors. The last figures show the speed of process migration.

System Throughput	KByte/S
Funnel Copy (**Same processor**)	11,299
Memory Copy (**Master to Local Slave**).	1,820
Process Migration (**Master to Local Slave**)	1,235
Process Migration (*Slave to Local Slave*)	1,045
Process Migration (**Master to Remote Master**)	794
Process Migration (**Master** to *Remote Slave*)	638
Process Migration (*Remote Slave to Remote Slave*)	530

Table 2: System data throughput (KByte/S)

In the above chart, 30MHz nodes are printed in boldface while 25MHz nodes are printed in italics. In every case, the involved masters were running at 30MHz. Note that a migration between two slaves not in the same workstation involves four processors: two masters and the slaves themselves. We note that the speed of data transfer between a processor and a common memory (on the VME bus) is expected to be around 3MByte/sec. As indicated earlier, the reliability of the network is essential, since in MOSIX the LAN is used for kernel-to-kernel communication. The results of Table 2 imply that the speed of process migration between different workstations (master to remote master) using the LAN is about 35% slower than the speed of local migration (master to local slave). This difference in migration speeds is expected to diminish when faster LANs become available. All of these results have ramifications in scaling MOSIX, since the different migration speeds are taken into account by the load-balancing mechanism when a process is considered for migration to a remote processor.

The next two tables show the relation between the granularity of the computation, the frequency of communication, and the speedup. Using a 6-processor configuration in 2 identical workstations, the computation included one master process assigning identical computational tasks to slave processes. Upon the completion of each task, the slave process communicates with the master and is assigned a new task. The amount of computation done in each iteration was increased from 0.1 to 10 seconds, resulting in a decrease in the communication and management overhead of the master process. In each case we allowed two different initial process assignment schemes. The first uses MOSIX load balancing, and the second uses static assignment by the master process. Note that in the first case, each process was executed for at least one second in the processor in which it was created before it was migrated. For comparison, we note that the total execution time of the benchmark when executed as a single process with no communication overhead was exactly 60 minutes.

In Table 3 we list the results of executing a master and six slave processes. The first column lists the amount of work assigned to each slave by the master process. Column 2 is the elapsed execution time for the whole benchmark. Column 3 lists the total overhead (user + system) time which resulted from the work distribution and the communication overhead. Column 4 is the speedup, defined as the total execution time using a single machine divided by the elapsed time using six processors. Utilization is defined as the ratio between the single-machine execution time and the measured overhead (column 3), divided by the product of the elapsed time and the number of processes. Note that utilization refers to CPU utilization for all work done (including overhead time), while speedup refers to actual work done (not including overhead).

	Automatic Process Migration				Static Process Assignment			
Work Unit *(sec)*	**Elapsed** *min:sec*	**Overhead** *user+sys (sec)*	**Speedup**	**Utiliz.**	**Elapsed** *min:sec*	**Overhead** *user+sys (sec)*	**Speedup**	**Utiliz.**
0.1	11:50	14.2+159.2	5.08	88%	11:38	12.0+157.2	5.16	90%
0.2	11:00	6.5+80.9	5.45	93%	10:50	9.5+80.1	5.54	94%
0.3	10:43	5.2+56.5	5.61	94%	10:34	5.0+55.8	5.68	96%
0.4	10:40	6.7+39.7	5.63	94%	10:26	5.7+40.7	5.75	97%
0.5	10:30	3.8+33.8	5.71	96%	10:22	5.0+32.9	5.80	97%
0.7	10:29	5.3+24.0	5.72	96%	10:16	2.3+25.6	5.84	98%
1.0	10:20	2.3+17.4	5.81	97%	10:12	2.4+18.0	5.89	98%
1.5	10:17	5.0+12.7	5.83	97%	10:09	3.1+13.0	5.92	98%
2.0	10:14	4.3+9.9	5.86	98%	10:08	3.8+10.3	5.94	99%
3.0	10:14	4.9+6.5	5.88	98%	10:06	3.2+6.9	5.94	99%
5.0	10:16	5.0+4.9	5.85	97%	10:06	4.4+4.7	5.95	99%
10.0	10:17	1.0+5.4	5.84	97%	10:04	1.0+9.3	5.96	99%

Table 3: Task granularity benchmark, master + 6 slave processes

Comparison of the task granularity shows that even with a small unit of work, the speedup and utilization obtained are quite good. Comparison of MOSIX load balancing and the optimal static assignment shows the efficiency of the MOSIX scheme. This is also reflected in the high degree of utilization obtained. The corresponding results for a master and five slave processes, each executing on a different processor are given in Table 4. Comparison between the two tables shows a consistent increase in the speedup and utilization ratios, with only 1-2% decrease in the utilization when the number of processes is increased from 6 to 7.

	Automatic Process Migration				Static Process Assignment			
Work Unit *(sec)*	**Elapsed** *min:sec*	**Overhead** *user+sys (sec)*	**Speedup**	**Utiliz.**	**Elapsed** *min:sec*	**Overhead** *user+sys (sec)*	**Speedup**	**Utiliz.**
0.1	13:49	6.6+127.5	4.35	90%	13:43	7.8+130.4	4.37	90%
0.2	12:57	1.2+67.5	4.63	94%	12:53	4.7+66.2	4.66	94%
0.3	12:41	4.7+41.6	4.74	95%	12:36	5.3+45.1	4.77	96%
0.4	12:35	3.7+35.3	4.77	96%	12:28	5.7+33.9	4.82	97%
0.5	12:27	2.7+29.1	4.82	97%	12:23	3.0+28.0	4.85	97%
0.7	12:22	3.2+19.5	4.86	97%	12:16	5.1+18.3	4.89	98%
1.0	12:19	2.0+16.9	4.88	97%	12:13	0.8+15.3	4.92	98%
1.5	12:23	4.2+10.2	4.85	97%	12:09	3.6+10.4	4.94	99%
2.0	12:13	4.8+7.9	4.91	98%	12:08	3.3+8.3	4.95	99%
3.0	12:13	4.2+7.0	4.92	98%	12:06	2.2+6.2	4.96	99%
5.0	12:15	2.4+5.3	4.90	98%	12:06	1.8+5.0	4.97	99%
10.0	12:15	1.0+8.4	4.90	98%	12:05	2.7+3.4	4.97	99%

Table 4: Task granularity benchmark, master + 5 slave processes

The last table shows how effectively some CPU-bound applications can utilize the load balancing algorithm of the system. A distributed implementation of the traveling-salesman problem was developed. It uses a master process which forks several subprocesses. The processes use System V messages for interprocess communication. The example shown is fairly large, requiring almost 40 minutes of CPU time. Each of the executions involves one master process and four slave processes.

Number of Processors	Total CPU Time	Elapsed Time	Speedup
1	2344.8	2348.4	1.00
2	2293.2	1178.6	1.99
3	2265.1	790.0	2.97
4	2264.8	598.1	3.93

Table 5: The traveling-salesman problem

The total CPU time given in Table 5 is the number of CPU seconds spent in user mode by all of the processes. As expected, this number is approximately the same regardless of the number of processors working on the problem. The elapsed time given is the real time required to execute the program. The speedups gained are quite impressive: all are linear. These speedups are obtained through the use of the automatic load-balancing mechanism of the MOSIX kernel.

6. CONCLUSIONS

In this paper we describe a symmetrical multiprocessor operating system which integrates a set of processors that share a common VME bus into a single-machine UNIX workstation. This system can further integrates several such workstations, via a local area communication network into a single-machine UNIX environment. With the present tuning, and the specific hardware described, the maximal configuration is 496 processors, configured into 62 workstations, with 8 processors each. Each workstation is expected to deliver a theoretical computing power of 64 MIPS, with as much as 8 MIPS for each process. Measurements carried out with a few 30MHz (10 MIPS) processors indicate that the system's performance will improve by 20 percent when all of the 25 MHz processors are replaced. Thus the expected total computing power of the maximal configuration is 4 - 5 GIPS. The cost/performance ratio of this workstation is expected to be $1,000/MIP. We note that the restriction on the number of workstations is imposed by the size of the UNIX process ID, which was preserved as a *short* integer. If this restriction were removed, then the number of workstations may be further increased.

As of the summer of 1988, a system of 3 workstations, each with 4 processors was operational. A workstation with 8 processors has also been tested. Our target configuration is 8 workstations, with 64 processors.

The software system is fully compatible with AT&T UNIX Version 5.2 including virtual demand paging for both local and remote processes (migrated to another processor) as well as the TCP/IP communication software. The development of the multiprocessor version kernel (from the uniprocessor version) required 4 man/months. In addition to the system kernel, an extensive set of software development tools and a powerful kernel debugger were also developed.

When used for a CPU-bound application, the system has shown a linear speedup with the number of processors. As can be expected, in I/O and communication-bound processes a lower speedup was obtained. In all cases, the load-balancing algorithm was a major contributor to the high utilization of the system when multiple processes were executed.

ACKNOWLEDGEMENT

The authors wish to thank A. Shiloh for the implementation, National Semiconductor Corp. for the technical support and G. Shwed for performing the granularity benchmarks.

REFERENCES

[1] *AT&T System V Interface Definition*, AT&T Ed., 1985.

[2] Bach M. J., *The Design of the UNIX Operating System*, Prentice-Hall, 1986.

[3] Barak A. and Litman A., "MOS: A Multicomputer Distributed Operating System," *Software Practice & Experience*, Vol. 15, No. 8, pp. 725-737, Aug. 1985.

[4] Barak A., Malki, D. and Wheeler R., "AFS, BFS, CFS... or Distributed File Systems for UNIX," *Proc. EUUG Conference on Dist. Systems*, pp. 461-472, Manchester, Sept. 1986.

[5] Barak A. and Paradise O. G., "MOS - Scaling Up UNIX," *Proc. Summer 1986 USENIX Conference*, pp. 414-418, Atlanta, GA, June 1986.

[6] Barak A. and Shiloh A., "A Distributed Load-balancing Policy for a Multicomputer," *Software Practice & Experience*, Vol. 15, No. 9, pp. 901-913, Sept. 1985.

[7] Cheriton D.R. and Zwaenepoel W., "The Distributed V Kernel and its Performance for Diskless Workstations," *Proc. of the Ninth Symposium on Operating Systems Principles*, pp. 129-140, Oct. 1983.

[8] Douglis F. and Ousterhout J., " Process Migration in the Sprite Operating System," *Proc. 7-th International Conf. on Distributed Computing Systems*, pp. 18-25, Berlin, Sept. 1987.

[9] Drezner Z. and Barak A., "An Asynchronous Algorithm for Scattering Information Between the Active Nodes of a Multicomputer System," *Journal of Parallel and Distributed Computing*, Vol. 3, No. 3, pp. 344-351, 1986.

[10] Morris J.H., Satyanarayanan M, Conner M.H., Howard J.H., Rosenthal D.S. and Smith F.D., "Andrew: A Distributed Personal Computing Environment," *Communication of the ACM*, Vol. 29, No. 3, March 1986.

[11] Ousterhout J., Cherenson A.R., Douglis F., Nelson M.N. and Welch B.B., "The Sprite Network Operating System," *IEEE Computers*, Vol. 21, No. 2, Feb. 1988.

[12] Popek G.J. and Walker B.J., *The LOCUS Distributed System Architecture*, MIT Press, 1985.

[13] Schroeder M.D., Birrell A.D. and Needham, R.M., "Experience with Grapevine: The Growth of a Distributed System," *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp. 3-23, Feb. 1984.

Sema: a Lint-like Tool for Analyzing Semaphore Usage in a Multithreaded UNIX Kernel

Joseph A. Korty
MODCOMP, an AEG company
1650 West McNab Road
Fort Lauderdale, FL 33340
305/977-1820
uunet!modcomp!joe

ABSTRACT

Sema is a tool that examines the use of semaphores in the source of a tightly coupled, multiprocessing UNIX[1] kernel. It provides a window into the use of semaphores in general, and into *cycles* (potential deadlocks) between semaphores in particular. This tool also ranks the semaphores so that new code can be added to a kernel in a deadlock free fashion.

This tool is not guaranteed to find all cycles. It can also report cycles where none exist. However, our experience at MODCOMP[2] with the 3DU2[3] kernel shows that both the kernel source and this tool can be tuned to improve the accuracy of the tool. The method of tuning is similar in spirit and practice to what is needed to make a program successfully pass *lint*.

INTRODUCTION

Sema is a tool which searches for deadlocks among the semaphores used to implement mutual exclusion in a multithreaded, multiprocessor UNIX kernel. It takes as its input the source of the kernel to be examined. Its output is a series of reports on the deadlocks that it believes are present. Upon examination of the reports, the developers of the kernel can then make *repairs* to eliminate each deadlock that proves to be real. For the deadlock reports which prove to be false, several ways are provided to *tune* the kernel and the tool so that the false reports can be eliminated in future runs, if that is desired.

Motivation for the tool

The development of MODCOMP's 3DU2 operating system involved the implementation and use of semaphores throughout every part of a more-or-less standard AT&T System V kernel. Since 3DU2 is targeted at the realtime marketplace, it was felt that every opportunity to maximize the robustness of the system before first shipment should be taken. Since extensive use of semaphores requires great care at preventing deadlock among those semaphores, it was felt that a tool which would search the source for such deadlocks would be a useful addition to the currently existing deadlock prevention methods.

1. *UNIX* is a registered trademark of AT&T in the U.S. and other countries
2. *MODCOMP* is a registered trademark of Modular Computer Systems, Inc.
3. *3DU2* is a trademark of Modular Computer Systems, Inc.

Overview

The next section presents some background information necessary for an understanding of the problem discussed in this paper. Following that are sections on the special conditions the kernel presents to semaphore analysis, how *sema* is used, implementation methodology, what reports are generated by *sema* and how they are to be interpreted, tuning, future directions, and of course, the ubiquitous conclusion section.

BACKGROUND

Readers familiar with multithreaded kernels, spin locks, semaphores and semaphore classes, semaphore graphs, cycles, and deadlocks, may want to skip this section.

Multithreading

The rising popularity of tightly coupled multiprocessors has led to the development of operating systems which can make effective use of the new hardware. The operating systems are almost invariably *multithreaded*; that is, they allow more than one processor/process pair to be concurrently executing system calls out of the same kernel. This facility is in response to the bottleneck on system calls that otherwise occurs around 2 to 3 processors under common workloads[1].

Multithreaded kernels are more vulnerable to data structure corruption from mutual exclusion problems than uniprocessor kernels are. Uniprocessor kernels need only to prevent interrupt level code and process level code from simultaneously modifying the same data structures. This is easily and economically accomplished by requiring that process level code lock out interrupts for the duration of an atomic update on a shared data structure.

Multithreaded kernels, by allowing multiple processes to execute kernel code, must also provide mutual exclusion between processes. This requires that many more data structures be protected. In addition, simply locking out interrupts is no longer enough; some other mechanism must also be used for mutual exclusion. Invariably, a combination of spin locks and semaphores are used.

Semaphores and spin locks

It is beyond the scope of this paper to present a complete discussion on semaphores, spin locks, and their uses; for that, see [1,4]. Rather, we will discuss only those features necessary for an understanding of this paper. It should perhaps be pointed out that the semaphores discussed in this paper are *not* the IPC semaphores System V makes available to user level code.

To request exclusive use of resource R, process A executes a *psema(semaphore)* or *spsema(spinlock)* on a semaphore or spin lock associated with R. Process A gets access to R if and only if no other process B already has such access. In the case of contention, A's request waits until B releases the resource.

The two semaphore services differ in that a spin lock, when refused access to R, will cause A to wait by wasting processor time, perhaps by spinning on a test-and-set instruction; while a semaphore will put A to sleep and allow some other process to continue. Each has its own uses. For example, spin locks are more efficient when the expected wait time is shorter than the time it takes to do a pair of context switches.

Whenever a process is finished with a resource, it releases the semaphore with the corresponding *vsema(semaphore)* or *svsema(spinlock)* service.

Semaphores and deadlock

It is well known that the use of semaphores can result in deadlock, and that the ordering of semaphores within the involved processes is the only attribute that determines if deadlock is possible or not. A simple example of deadlock, one which also illustrates an important notation used here and in an expanded form in other papers, is given below:

```
A//p(x),p(y),v(y),v(x)
B//p(y),p(x),v(x),v(y)
```

This defines two processes, A and B, as consisting of a sequence of semaphore and/or spin lock operations (we usually don't care which when looking for deadlocks). A $p(x)$ makes a request for semaphore x, and a $v(x)$ releases it. In the above example, deadlock between A and B results whenever both processes get to complete their first semaphore request before their second request is initiated.

Deadlock can be observed more easily by building up a *semaphore graph*. This is a directed graph whose nodes are the semaphores (or, more properly, the *semaphore classes*, defined below) in the kernel, and whose vectors connect all pairs of semaphores which the kernel locks concurrently. Each vector is directed from the first to the second semaphore that is locked.

If the semaphore graph is acyclic, then the set of processes is deadlock free. If it is cyclic, then any deadlocks present are modeled by the cycles, but not all the cycles displayed are deadlocks. For example:

```
A//p(w),p(x),p(y),...
B//p(w),p(y),p(x),...
```

As long as concurrent requests for x and y operate under the shadow of w, no deadlock is possible, even though x and y are themselves involved in a cycle. Such cycles are said to be *unreachable* in the deadlock literature.

Still, semaphore graphs remain useful. We know that if we can make the graph acyclic, then the set of processes will be deadlock free. One easy way to accomplish this is due to Havender[5]. Havender's method assigns a rank to each semaphore. All processes that need to simultaneously hold two or more of the semaphores must request them in ascending order, in accordance to this ranking.

Two other methods of detecting deadlocks are *process state diagrams*[6] and *process graphs*[2]. Both methods appear to be restricted to a fixed set of processes. This limitation does not apply to semaphore graphs. As will be seen later, kernel deadlock analysis requires a technique that will work with an infinite number of processes.

Another popular method for resolving deadlocks is the use of the conditional semaphore request, *cpsema*. When two semaphores must be locked out of order, then the second is locked with *cpsema*. If the resource is available, *cpsema* grants it; but if the resource isn't available, then *cpsema* returns a failure code back to the requesting process. It is then up to the process to initiate whatever recovery procedures that are applicable.

Semaphore classes

Occasionally it is more appropriate to group semaphores into *semaphore classes*, and graph the classes instead. This happens, for example, when many identical instances of a resource (such as memory pages in a virtual memory implementation) are each protected by its own semaphore. Such semaphores often operate under the shadow of a protecting semaphore, which is the one actually used to prevent deadlock with the rest of the semaphores in the system.

SPECIAL CONDITIONS IN THE KERNEL

Threads

The kernel, of course, is not a process in the traditional sense of the term. Rather, it is a single body of code through which an unbounded number of threads of execution may be running. These threads, and not UNIX processes, are what correspond closely to the process concept of the deadlock literature. More formally, we define a *thread* as one of the many valid sequential paths through the kernel.

There are two types of threads in a kernel: process threads and interrupt threads. A *process thread* starts when a UNIX process initiates a system call, and exits when that system call exits. An *interrupt thread* starts whenever the processing of an interrupt begins, and ends when the processing of that interrupt ends. A thread thus represents a kind of "virtual machine", which is created whenever the service represented by the thread is needed, and which is destroyed at the time the service completes.

Threads differ from the process concept of the deadlock literature in one important respect: an unbounded number of instances of each thread may be executing at any time, while the process model assumes only one instance is running at a time. This may happen, for example, when two UNIX processes simultaneously execute the same system call. This important distinction means that a thread can, in effect, deadlock with itself. For example,

```
A//p(x),p(y),v(y),v(x) ... p(y),p(x) ...
```

is perfectly safe under the process model, but runs the risk of deadlock under the thread model.

Cycles

The semaphore literature mentions several methods for detecting deadlock. Perhaps the most prominent is the process graph model, reputedly invented by Dijkstra[3], and recently refined by Carson[2] to support the *n*-process model. A less prominent method is the analysis of cycles in a semaphore graph. Cycle analysis is defective in that it can report deadlock when in fact there is none. It can also be used only on binary semaphores (the only kind that we have been discussing). Process graphs give an exact solution.

Sema is based on cycle analysis. There are several reasons for this. First, cycle analysis is the only method currently known that is not restricted to a finite number of threads, or instances of such. Second, an important function of *sema* is to give its users a good idea of why each deadlock exists, where it is occurring in the source of the program, and what can be done about it. A simple go/nogo report is not sufficient. Although the process graph model has some predictive power, it was not clear how to get it to produce the kind of detailed report that the cycle method is able to produce naturally.

INVOCATION

Sema is invoked in a manner similar to *lint*. *Sema* differs principally by not allowing incremental processing, that is, all the source must be given to the tool at the same time.

The same **-D** and **-I** options used in a real compilation should be given to *sema*. These are the only options accepted by the tool.

Sema handles only *.c and *.h files. Any *.s files which are not simple leaf routines or use semaphores themselves should be modeled in a stub .c file.

Sema can be customized by two user-definable filters, skeletons of which are provided with the tool. Customization will be discussed in more detail in the section on tuning.

Sema outputs a series of reports. Like *lint*, a concatenation of the reports goes on *stdout*. Unlike *lint*, each report is also placed into its own output file. These reports are discussed in more detail later.

The following example is typical of actual use. Notice that only the major source files of the kernel are processed. By convention, as much tuning as possible is concentrated into the two custom filters and into *stubs.c*, a "program" which is never compiled into a real kernel.

```
sema -DINKERNEL -I. os/*.c fs/s5/*.c io/*.c stubs.c
```

IMPLEMENTATION

Sema is implemented in three passes. Pass 1 extracts the threads from the kernel source. Pass 2 generates the semaphore graph, with attached debug information, from the threads. Pass 3 does all the reports. The first two passes are discussed in this section; pass 3 rates its own section.

The output of all passes is in human readable form. They provide, in their own right, an interesting but not particularly relevant window into the kernel.

Extraction of threads from the source

As defined earlier, a thread is some valid sequential path through the kernel. Thus, every *if* statement, every loop construct, every *switch* statement (and so on) encountered in the source could potentially fork off a new set of threads, if those statements meet the right conditions. A combinatorial explosion of threads occurs as all possible combinations of these statements are considered. Clearly, some method of pruning this set of threads (*thread reduction*), or managing them in a compact manner (*thread compaction*), is required.

Thread compaction can be achieved with a *semaphore data flow graph.* This idea, borrowed from compiler technology, breaks up the input source into basic blocks, connected by flow control statements. Not every basic block and flow control statement appearing in the source will appear in the graph, however; just those contributing to the sequence of semaphore operations that make up the threads, or needed for semaphore debugging purposes, will appear.

Although data flow graphs can represent an infinite number of threads, they do, in fact, have one defect. They model every possible path through the kernel, but not every such path is a thread. For example,

```
if(ttydevice) psema(ttysema);
else psema(lpsema);
     :
<other statements>                    (1)
     :
if(ttydevice) vsema(ttysema);
else vsema(lpsema);
```

represents two unique thread fragments, not four.

Thread reduction is another powerful strategy, one that discards threads or merges significant fragments of threads until the number of threads have been reduced to a more manageable level. The idea is based on the observation that many of the threads and thread fragments present in the kernel are redundant as far as deadlock analysis is concerned. Thread reduction should be done in such a way as to minimize or eliminate the damage to threads, from a deadlock detection point of view.

call graphs

Sema implements a simple extraction algorithm that does both thread compaction and thread reduction. It is the *call graph*, which is simply a list of the procedure definitions and procedure calls that are defined in the source, in the order that they appear in the source. The call graph is the output of the first pass, and its format is as shown in the following call graph fragment.

```
xopen
  xname
  xopen1
xpage
  p(mlock)
  xfree
  v(mlock)
```

This example defines two procedures, *xopen* and *xpage*. The source of *xopen* contains calls to two procedures, *xname* and *xopen1*, in that order. A similar statement can be made about procedure *xpage*, except that two of its "procedure calls" are semaphore operators. A semaphore operator is unique among procedure calls, in that it has the name of the semaphore operated upon passed as an argument.

The call graph model merges and reduces all the true threads to a handful of virtual threads, each of which is to be searched for deadlock. The root of each virtual thread is any procedure definition that doesn't have a *parent*; that is, it is not called by any other procedure. All entry points to interrupt driven code automatically have no parents, and hence each form the root of a thread. The same is true for the entry points to each of the system calls.

call graph accuracy

The call graph model is such a simple method, and performs such strong transformations on the source, that it is difficult to believe that it can be generally safe. But thanks to modern coding practices, it often is. Some constructs, such as *goto* and *longjmp*, can introduce significant errors into the model, but modern coding practice usually makes their appearance in the source rare. In fact, most uses of *goto* tend to do a simple escape out of deeply nested constructs, ala *break*. When this is the case, the error contribution to semaphore analysis is no worse than that of *break*.

A similar argument can be made for loops. Most loops on each iteration release whatever semaphores were locked on that iteration, and no others. Thus, such loops can be accurately modeled by replacing the loop with a single iteration of it.

If and *switch* statements present a greater source of error in the model. For example, reducing (1) to a call graph results in the following call graph fragment:

```
p(ttysema)
p(lpsema)
v(ttysema)
v(lpsema)
```

which erroneously states that there is a relationship between *ttysema* and *lpsema*. However, errors of this type are more acceptable than missed semaphore relationships, as all they usually do is report nonexistent deadlocks. This is acceptable, given the enormous reduction in threads achieved by the method, and in light of the *lint*-like nature of the tool.

A more serious failure occurs from an interaction of *if* statements and loops which violates the assumption that each iteration of the loop locks and releases the same set of semaphores. For example,

```
for(i=0; i<limit; i++) {
  if(i==(limit/4)) psema(looplock);
    :
  <other lock, unlock statements>
    :
  if(i==(limit/2)) vsema(looplock);
}
```

Such loops can be more accurately modeled by unrolling each unique iteration. However, this is not done in *sema*.

The most fundamental problem with semaphore analysis from source has to do with accurate *semaphore identification*. Semaphore identification is easy as long as the actual names of the semaphores are used in every semaphore call (as it has been in all the examples given so far). But identification becomes difficult when, for example, pointers to semaphores and arrays of semaphores indexed by arbitrary expressions are used. There is no easy way to solve this problem when working from source; however, global data flow analysis can solve some of the simpler cases. *Sema*

does not apply any such automated techniques. However, it does provide a manual tuning phase which, when properly used, greatly increases the accuracy of the tool. This technique is described later, in the section titled *Tuning*.

Extraction of the semaphore graph from the call graph

output generated

Pass 2, the semaphore graph, is created from the threads embodied in the call graph produced by pass 1. This output is a text file, and consists of a list of the vectors making up the semaphore graph. Each vector is called a *semaphore pair*, due to the prominence of the two semaphore nodes which make up the end points of the vector.

In principle, only the names of the semaphores that make up each pair needs to be output. But in reality, ancillary information has to be attached to each vector, because the purpose of *sema* is not just to detect deadlock, but to help the user to discover its causes and its cure. Because of this ancillary information, several copies of the same vector, each with different ancillary information, may be output. The manner in which this information is useful will be explained in the section on report generation. For now, let us look at a fragment of the pass 2 output:

```
x|pio      y|freg      /undo/
x|pio      y|useg      /
y|mres     z|swp.c     ~mallo~unswp/
```

Each line identifies two semaphores which are being locked (*x*, *y*, *z*), the procedures which locked them (|*procname*), the call/return path taken between the two procedures (the "/undo/" column), and any relevant flag information (the ".c" field). How to read all this is best described by translating the third line of the above into English:

> "Semaphore *y* was locked in procedure *mres*. While that lock was held, *mres* returned to its caller (~*mallo*), which in turn returned to its caller (~*unswp*). *Unswp* then called *swp*, which in turn locked semaphore *z* with the *cpsema* (the ".c" flag) service."

The path field identifies the trail of procedure calls and returns, if any, taken by the thread in order to get the two semaphores to be simultaneously locked. Note that the procedures in which the locks occurred are not considered to be part of the path. Also, any *side jaunts* (a series of calls and matching returns) taken by the thread between the two semaphores are not relevant to any analysis of the semaphore pair, and so are stripped from the path.

Any single character flags that are present are attached to the corresponding *procname* by a separating period. Currently, the only two flags characters are "c" (for *cpsema*) and "s" (for *spsema*).

The purpose of the pass 2 syntax is to organize the data in such a way such that the most important information stands out, and the less important information appears successively in less prominent places. The most important is the names of the semaphores that make up the vector and the names of the procedures in which they were locked. Of less importance is the path taken between those two procedures, and of still lesser importance are any attached flags.

extraction algorithm

The algorithm first finds the root procedures on the call graph. Then, it walks the tree of procedure calls, maintaining a list of locked semaphores it has encountered on the way. A *p()* adds a semaphore to the list, and a *v()* deletes a semaphore from the list. Whenever a semaphore is locked, *n* vectors are generated, where *n* is the number of semaphores already present on the locked list. Each generated vector describes the relationship between the newly locked semaphore and one of the *n* semaphores already on the locked list.

The path information is created by maintaining a stack of procedure names for each semaphore on the locked list. Each time a call graph procedure is entered into or returned from, appropriate

adjustments are made to each of the stacks.

A filter embedded in the output eliminates those output lines considered to be redundant. Our experience has shown that if multiple paths exist between the same two uses of a semaphore, only one of them (the shortest) is needed in order to make an accurate judgement of its location and the cause of its existence. For our kernel, this filter reduces the pass 2 output twentyfold.

Recursion is rare in the 3DU2 kernel. When encountered, the tree is pruned in the usual way of tree walking algorithms.

GENERATION OF REPORTS

The 2-semaphore cycle report

The semaphore graph may or may not contain cycles among the semaphores. If there are no cycles, then the graph is acyclic, and no cycle report is made. If there are cycles, then anywhere from two to all of the semaphores will be involved in each cycle. This section just covers cycles involving two semaphores.

A cycle display is a subset of the pass 2 output. It displays all occurrences of the use of the pair of semaphores that are involved in the cycle. For example,

```
CYCLE:
r|ex          f|up.c     /dtach/...
r|done        f|up.c     /dtach/...
r|lreg        f|up.c     ~alloc/...
r|pdup        f|up.c     /dreg/...
r|pxmt        f|up.c     /xdup/...
r|handy       f|up.c     /getit/...
f|up.c        r|rel.c    /writeit/...
```

In this example, semaphores *r* and *f* are concurrently locked at seven "places" in the kernel. In all but the last, *r* is locked first. This hints that, if deadlock is present, the sole instance in which *f* is locked first is most likely to be the culprit. The flags field, however, shows that the author of procedure *rel* was careful enough when locking *r* in the reverse direction to use the *cpsema* (rather than the *psema*) service on it. This hints that the author was aware of the reverse locking, and has probably compensated for it already. A quick check of the source will reveal if this is the case.

This example shows the suggestive power of this report. By listing all interesting occurrences of the semaphore pair in the source, the path that has be taken to cause the creation of that pair, and how each semaphore was used, the user automatically gets a powerful window into the causes, location, and cure of each cycle.

The n-semaphore cycle report

Cycles of three or more semaphores are reported only as a side effect of the topological sort used to create the acyclic reports discussed below. The report only names the semaphores involved in each cycle. Clearly, more work is needed in this area.

The acyclic level report

The level report assigns a numerical value or *rank* to each semaphore. This value is such that, for every semaphore pair (x,y) not involved in a cycle, $rank(x) < rank(y)$. For semaphore pairs that are involved in a 2-semaphore cycle, automatic *cycle breaking* is applied so that these semaphores can also be assigned a ranking.

The primary purpose of the ranking is to permit fresh code to be added to the kernel in a deadlock free manner via Havender's method: as long as multiple locks are requested and held in an ascending order, then deadlock is not possible. If they must be requested out of order, then other means to prevent deadlock have to be applied by the kernel developer.

The level report can accurately assign levels only when the input semaphore graph is acyclic. If it is cyclic, then the cycles have to be implicitly or explicitly broken in order to get the report. If a cycle is broken the wrong way, then an error occurs in the ranking of the involved semaphores.

The level report applies two types of automated cycle breaking. In the first, *cycle weighing*, each cycle reported in the 2-semaphore cycle report is examined. Whichever locking direction occurs the least is the one that is removed from the semaphore graph. If both directions occur equally often then both are removed.

The second type breaks all cycles involving three or more semaphores. This happens automatically as a result of the topological sorting algorithm used to assign the levels. These cycles are broken arbitrarily, without any consideration as to the "best" spot to apply the break.

The acyclic parent/child report

Although in principle the level report correctly states the locking order between two semaphores when there is a relationship between them, it also incorrectly implies a locking order when there is no relationship. A more fine-grained display of the topological sort would be useful. This is provided in the *parent/child report*. For every semaphore, this report lists every parent (semaphores in which this is the second of a pair) and every child (semaphores for which this is the first of a pair). The associated level assignment is displayed alongside the name of each semaphore.

TUNING

Sema's accuracy is noticeably increased if the kernel source and the tool itself are tuned. Source tuning can be accomplished by several mechanisms. For small tunings within a *.c file, an *#ifdef SEMATRACE* statement can be used to modify the source. (This is similar in spirit to the *lint* macro definition used by *lint*). For larger tunings, such as modeling *.s files, stub *.c files can be written which model the required behavior, from *sema*'s point of view.

Sema is provided with two custom filters. The first, *semacustom*, is applied to the *cpp* output. It currently maps all indirect procedure calls (e.g., file system switch, cdev switch) into a list of procedures callable at those points. But in fact, it can be used to modify any aspect of the input C code before it is reduced to a call graph.

The second filter, *semacustom2*, makes any desired modification to the call graph before the call graph is made available to the second pass. Currently, it is used to delete semaphores that are not used for mutual exclusion purposes, and to rename semaphores that belong to a common semaphore class to a common name.

EXPERIENCE

Sema was used to find and resolve deadlock on the 3DU2 operating system while both were in the early phases of development, and, unfortunately, no statistics were gathered during those hectic days. So these results are skewed a bit, since they come from a more mature phase of the development of the operating system.

On the latest run 36 2-cycles and 5 n-cycles, $n>2$, were reported. It took one man-day to examine the 36 cycles, determine that two were deadlocks, and correct them. The five n-cycles have not yet been examined. The tool took about 3 hours, on a 25Mhz 68030, to process about 100,000 lines of C code. The code had about 1,100 procedure definitions, 75 semaphore classes grouped into 14 levels, and 130 nontrivial compacted threads.

FUTURE DIRECTIONS

Sema is quite limited, compared to what it could do. This is typical, after all, of first attempts; the struggle to get it all together generates many new ideas on how to do it better. Many of the improvements have already been discussed in the text. This section summarizes those, and introduces a few new ones.

- Add an n-semaphore cycle report, $n>2$.
- Use a flow graph representation of threads.
- Bypass graph extraction from source entirely. Instead, bring up a version of the kernel which has thread or semaphore pair logging capability built into its profiler. A set of applications could then be tailored to exercise many of the threads in the kernel.
- An automatic cycle breaking algorithm which takes a more comprehensive view in deciding how a cycle is to be broken.
- Automatic recognition and removal of semaphores not used for mutual exclusion purposes.
- Automatically trace indirect procedure calls whenever possible.
- Replacement of the custom filters with a command language.
- Support deadlock analysis in sets of ordinary unithreaded processes.
- Less cryptic output.

Many of these improvements are necessary to convert *sema* into a general purpose tool, one that can be applied effortlessly by ordinary users to their semaphored programs.

CONCLUSIONS

Sema is a powerful deadlock tool that works directly from the source. It accounts for the fact that real processes are not invariant straight line sequences of semaphore statements, but instead dynamically change these sequences from run to run due to the flow control statements of the process. In addition, its ability to pinpoint where in the source each deadlock resides, and give just the information needed by the user to decide on its cause and cure, is an important accomplishment of this project.

Our current implementation falls short in several areas. Only monolithic, multithreaded applications can be processed. Several useful features, which are easy to implement, are not present. And some fundamentally unsolvable problems could be mitigated further than has been done so far.

Because it is not perfect, *sema* is most useful as a confidence check on whether or not our other techniques for avoiding deadlock in the kernel really worked. In any case, it was not envisioned as way to ignore the possibility of deadlocks at the beginning of kernel development, and then fixing up the problems later. Such a tool would have to be much more comprehensive than *sema* is.

Still, the tool has proven to be useful. It has found several deadlocks for us, and has provided greater insight into the global behavior of semaphores in the kernel. This in turn has led to the serendipitous discovery of several semaphore *windowing problems* (incomplete mutual exclusion protection), whose existence otherwise might have escaped us.

REFERENCES

[1] Maurice J. Bach and Steven J. Burnoff, "Multiprocessor UNIX Operating Systems", *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, October 1984, pp. 1733-1749.

[2] Scott D. Carson and Paul F. Reynolds, Jr., "The Geometry of Semaphore Programs", *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 1, January 1987, pp. 25-53.

[3] E. G. Coffman, Jr., M. J. Elphick, and A. Shoshani, "System Deadlocks", *ACM Computing Surveys*, vol. 3, no. 2, June 1971, pp. 67-78.

[4] E. W. Dijkstra, "Co-operating Sequential Processes", *Programming Languages*, F. Genuys (ed), Academic Press, New York, 1968, pp. 43-110.

[5] J. W. Havender, "Avoiding deadlock in multitasking systems", *IBM Systems Journal* 2 (1968), 74-84.

[6] Richard C. Holt, "Some Deadlock Properties of Computer Systems", *ACM Computing Surveys*, vol. 4, no. 3, September 1972, pp. 179-196.

ACKNOWLEDGEMENTS

I wish to thank Hagai Ohel for several discussions which had a profound influence on the project, Jim Houston, who suggested this might be something worth writing about, and Meg McRoberts, who lent her editorial expertise. Thanks also to all the 3DU2 staff members who took the time to proofread the paper. Special thanks goes to Dan Grostick and Dr. Guy Rabbat, for providing the exciting environment here at MODCOMP that has led to this and other works.

Visualizing X11 Clients

David Lemke
David S. H. Rosenthal

Sun Microsystems
2550 Garcia Ave.
Mountain View CA 94043

ABSTRACT

The color model of version 11 of the X Window System exposes a number of aspects of the underlying display. The concept it uses to represent these device-dependencies is the *Visual*; there may be several Visuals available on a given display. We present example programs to illustrate the techniques required if an X client is to operate correctly across the entire range of possible combinations of Visuals.

"The polymorphic visions of the eyes and the spirit are contained in uniform lines of small or capital letters, periods, commas, parentheses – pages of signs packed as closely together as grains of sand representing the many-colored spectacle of the world on a surface that is always the same and always different, like dunes shifted by the desert wind."

Italo Calvino, *Six Memos for the Next Millennium*

1. Introduction

A design objective of Version 11 of the X Window System†[6] is to make it possible to write client programs that operate correctly across a wide range of displays. This objective is often referred to as "device-independence", but this is something of a misnomer. X11 abstracts the properties of popular display types into an object called a *Visual*, and exposes to the client the existence of a number of Visuals. Visuals are divided into classes; each class representing the abstraction of a particular type of display hardware.

The implementor of an X11 server for a particular display must decide on the appropriate Visual classes to export to its clients. More than one Visual per screen may be exported to allow access to the full set of hardware capabilities. When a client connects to an X11 server, it must determine the Visuals the server is exporting and their classes, adapting its behavior to suit. Normally, this will involve the client choosing one of the Visuals, and changing appropriate initializations.

Simply adhering to the X11 protocol does not ensure that a client will operate correctly on all possible combinations of Visuals. The early X11 server implementations that were made available exported only a single Visual per screen, and these Visuals were of only two classes. As a result, many existing X11 clients operate correctly on single-Visual screens of these two classes, but not on others.

† The X Window System is a trademark of the Massachusetts Institute of Technology.

Among these not-really-device-independent clients are the **xwd** and **xwud** programs in the X11R3 distribution. They "dump" to a file, and "undump" from a file, the image of a window, but they don't work across all combinations of Visuals. In particular, they don't work:

- for TrueColor or DirectColor Visuals.
- if there are subwindows that have a different depth, Visual, or colormap.
- if the default colormap of the screen being dumped to doesn't mimic the colormap of the window being dumped.

We first review the Visual concept, and then present versions of the screen dump and undump clients that do work correctly across all Visuals. These clients are designed as teaching aids; they illustrate the techniques needed to make truly device-independent X11 clients, but they:

- are as simple as possible, and thus too inefficient for production use,
- ignore all the issues of interacting with window managers and other clients,[3] in the interests of clarity,
- and ignore our own advice about using a toolkit,[4] in order to focus on the underlying X11 mechanisms.

2. Visual Classes

An X client paints in a *Drawable*, a rectangular array of numbers called pixel values, using operations that alter the numbers, such as CopyArea and PolyLine. The only difference that affects the drawing process between one drawable and another is the *depth*, the number of bits in each of the numbers. The Visual concept is irrelevant to this process; a useful model is to think of CPU access to a display bitmap, with the monitor turned off (see Figure 1).

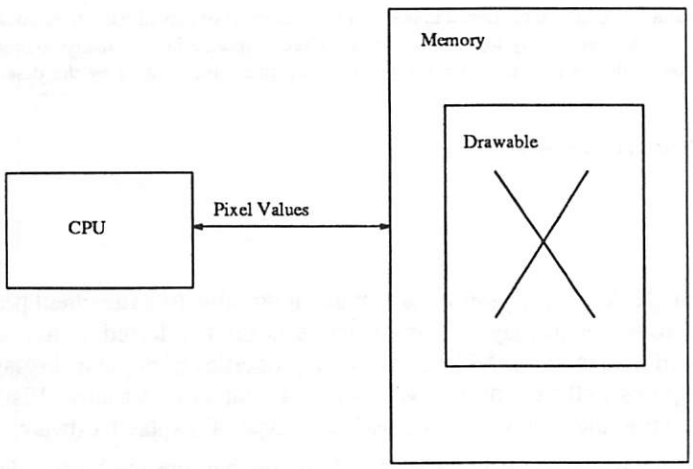


Figure 1: The X11 Drawing Process

Some Drawables are *Windows*, and these can be made visible by *mapping* them to the screen. When the client creates a Window, it must be assigned one of the possible Visuals for the screen; this Visual will control the process by which the Window's pixels become visible when it is mapped. A useful model is to think of video refresh access to a display bitmap, with the CPU halted (see Figure 2).

Conceptually, when a screen pixel is to be refreshed the top-most Window's corresponding pixel value is read, and used to index into a *Colormap*, an object containing three arrays (red, green and blue) of intensity values that models a video look-up table. The Visual for the window controls:

- How this indexing is done.
- Which of possibly many installed Colormaps is used.
- Whether and how the Colormap can be modified.

Visuals are divided into six classes, modelling six different types of display hardware. The classes, and their effects on the pixel value to visible color mapping, are:

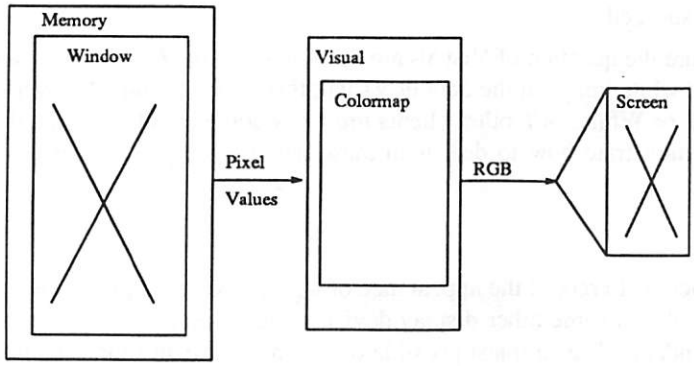


Figure 2: The X11 Display Process

- *StaticGray*. The pixel value indexes a predefined, read-only colormap. For each colormap cell, the red, green and blue values are the same, producing a gray image.
- *StaticColor*. The pixel values indexes a predefined, read-only colormap. The red, green and blue values for each cell are server-dependent.
- *TrueColor*. The pixel value is divided into sub-fields for red, green and blue. Each sub-field separately indexes the appropriate primary of a predefined, read-only colormap. The red, green and blue values for each cell are server-dependent, and are selected to provide a nearly linear increasing ramp.
- *GrayScale*. The pixel value indexes a colormap which the client can alter, subject to the restriction that the red, green and blue values of each cell must always be the same, producing a gray image.
- *PseudoColor*. The pixel value indexes a colormap which the client can alter. The red, green and blue values of each cell can be selected arbitrarily.
- *DirectColor*. The pixel value is divided into sub-fields for red, green and blue. Each sub-field separately indexes the appropriate primary of a colormap that the client can alter.

The early X11 server implementations provided two Visuals:

- Servers for monochrome displays exported a single StaticGray Visual of depth 1.
- Servers for color displays exported a single PseudoColor Visual of varying depth, typically 4 or 8.

These are the natural choices for simple workstation hardware, but they made it easy for initial clients to ignore many of the problems of dealing with Visuals.

Already, servers for common hardware such as the MIT sample server for the DEC QDSS display and the X11/NeWS[5] server on Sun color hardware are exporting multiple Visuals. The QDSS server's default Visual is PseudoColor, but it also exports a StaticColor Visual. The X11/NeWS server's default is a StaticColor Visual, and it also exports a PseudoColor Visual. As X11 becomes available on more complex hardware, multiple Visuals will become increasingly common.

Many current X11 clients make unwarranted assumptions about their Visuals. Among the most common such assumptions† are:

- That the default Visual is the only available Visual. Clients making this assumption may fail even though a Visual that could have supported them was available.
- That all Visuals with more than two Colormap cells are color. Clients making this assumption will behave strangely on some StaticGray and GrayScale Visuals.
- That all Visuals with more than two Colormap cells have writable Colormaps. Clients making this (very common) assumption will fail with Xlib errors on StaticColor and TrueColor Visuals.

† The technical term for these erroneous assumptions is *bugs*.

- That Colormaps (and especially the default Colormap) are infinitely large, so that attempts to allocate private cells in them will always succeed.

In fact, the only clients that can ignore the question of Visuals are those that use the *BlackPixel()* and *WhitePixel()* macros to paint a black and white image in the default Visual; this will work on all servers though the colors may not actually be Black or White. *All* other clients must pay some attention to the details of the Visual(s) they are using. To demonstrate how to deal with these details, we present versions of the screen dump and undump clients.

3. The "dump" Client

The objective of the screen dump client is to record the appearance of a specified rectangle of the screen so that it can be restored later, most likely on some other display device. Thus, the stored representation of the rectangle must be device-independent. The simplest possible device-independent representation is the red, green and blue (RGB) intensities of each pixel.

The only X11 protocol request that allows a client to read back the contents of a window is GetImage, so the dump client clearly must be based on it. From the point of view of the screen dump client, there are two problems with the specification of GetImage:

- It returns pixel values, not RGB values. Pixel values are not device-independent, and so cannot be used directly.
- Some of the values it returns may be garbage.

3.1. Converting pixel values to RGB

To transform the returned pixel values to RGB intensities, the client must use the QueryColors request to look up the index in a specified Colormap. The Colormap to use will be given by the colormap attribute of the window in question.

The Visual to be used for this lookup is implicit. Since setting a window's colormap attribute to a Colormap that does not match the Visual of the window is an error, using the Colormap attribute of the window for the pixel value lookup implies using the correct Visual.

3.2. Obtaining Correct Data from GetImage

The problem of GetImage returning garbage is described in the X11 protocol specification:

> "If the window has a backing store, then the backing-store contents are returned for regions of the window that are obscured by noninferior windows; otherwise, the returned contents of such obscured regions are undefined. Also undefined are the returned contents of visible regions of inferiors of different depth than the specified window."[7]

There are therefore two reasons why some of the returned pixel values may be wrong:

- They may be obscured by non-inferior windows, and the server may not be maintaining backing-store for the window. The backing-store attribute of a window is a hint to the server; there is no way for a client to discover whether the server is actually maintaining backing-store for any particular window at any particular time.

  Thus, unless it can be determined that a pixel in a window is *not* obscured by a non-inferior window, that pixel value must be regarded as garbage.
- They may be in a visible region of an inferior of different depth.

  Thus, unless the pixel in question can be shown *not* to be in an inferior of different depth, its value must be regarded as garbage.

In both these cases, at some cost, we can:

- unambiguously determine that a pixel value is good,
- assume that all others are garbage,
- and use GetImage on other windows to discover the correct values for these assumed-to-be-garbage pixels.

Unfortunately, there is a third reason why the pixels may be garbage:

> "When no valid contents are available for regions of a window and the regions are either visible or the server is maintaining backing store, the server automatically tiles the regions with the window's background unless the window has a background of **None**. If the background is **None**, the previous screen contents from other windows of the same depth as the window are simply left in place if the contents come from the parent of the window or an inferior of the parent; otherwise, the initial contents of the exposed regions are undefined."[7]

In other words, the pixels in areas whose top window has background **None** may not have been overwritten by any pixels drawn to the top window, they may simply have been left there when the window was mapped or exposed.

There is no way for a client to determine which pixels of a window have been updated by operations to that window (or its children). Thus there is no way to discover which pixels of a top window with background **None** are garbage, and even if there were there is no way to discover the correct pixel value or how to interpret it. The screen dump client simply has to live with the problem; its results cannot be guaranteed correct if any visible windows have background **None**.

There is one further problem. The protocol specification describes how a Pixmap or a single pixel value can be used as the border for a window. It does not specify which Visual and which Colormap are to be used to display these values. It appears that this is an oversight, and that the Visual and the Colormap of the window are to be used to display its border. The dump client assumes this.

3.3. Dump Algorithm

We use the painter's algorithm, creating an array of pixels to hold the output, and painting the windows into it from the back forwards.

Each pixel in the output may be painted a number of times, but the last time it is painted will be by the window on top at that pixel. Since this last pixel painted is not overlaid, we are assured that the pixels returned by GetImage will be good (subject to the background **None** problem).

Here is an outline of the algorithm; the code itself is shown in Appendix A:

- Create a data structure with one entry per pixel in the specified rectangle, and an empty list whose entries represent Colormaps in use. Each entry in the list is itself the head of a list of pixel values in use with that Colormap.
- GrabServer to prevent changes to the window tree while all this is going on.
- Start at the root, and:
  - Use GetGeometry and GetWindowAttributes to find the details of the current window.
  - If the current window is Viewable:
    - If the window's Colormap isn't in the list, create an entry describing it, and add it to the list.
    - Use GetImage on the current window, and for every pixel in the returned array:
      - Label every corresponding pixel in the data structure with the pixel value returned by GetImage, and with this window's Colormap.
      - If this is the first time we've seen this pixel value for this Colormap, use QueryColor to discover the RGB values that this window's Colormap maps the pixel value into, and store the pixel value and RGB in the list of pixel values in use for this Colormap.
    - Use QueryTree to find the list of children of the current window and recurse, starting with the head (bottom-most) of the list.
- UngrabServer, we're finished with it.
- Write to the output the width and height of the rectangle, and the number of pixel values we've added to the Colormap lists (this is an upper bound on the number of the distinct colors in the output).
- For each pixel in the data structure we built:
  - Find the list entry for the Colormap the pixel is labelled with.
  - In the list of pixel values in use for that Colormap, find the pixel value the pixel is labelled with.

- Write to the output the RGB value stored for its pixel value in the list of pixel values in use for the Colormap.

3.4. Assessment of the Algorithm

This algorithm is about as simple as it can be while still getting the job done, but it causes much more protocol traffic than is strictly required.

By making the very pessimistic assumption that a pixel overlaid by *any* other window is potentially garbage, it asks the server to return the value of each of these pixels more often than is essential. A more efficient algorithm would only regard pixels overlaid by non-inferiors or inferiors of different depth as potential garbage.

Space consumption may also be a problem. There are potentially up to 2^{32} different pixel values, and we must retain a list of each pixel value in use for every Colormap. Fortunately, the list of values in use cannot get longer than the $[w \times h]$ of the specified rectangle (in the case where every pixel in the rectangle is different), and will normally be much less.

4. The "undump" Client

The objective of the screen undump client is to restore the appearance of the rectangle of the screen that was dumped earlier. It may not in fact be possible to do this, since the Visuals available to the undump program may not be capable of displaying all the colors in the original image.

The undump client has two main tasks:

- Choose an appropriate Visual and create a Colormap for it.
- Build an image in memory in which the RGB values have been converted to pixel values using the Colormap.

For each of these tasks, we review the general problem and then focus on the specific features needed for the undump client.

4.1. Choosing a Visual

The server, during the connection handshake, provides the following information for each Visual on each screen:

- The depth.
- The Visual class.
- The masks that identify the red, green and blue sub-fields of the pixel value.
- The size (number of cells) of the Colormaps of the Visual.
- The number of bits in a red, green or blue value that the server will regard as significant.

There can be no general rules for choosing a Visual; a particular application might regard any of these pieces of information as the most important factor. For example:

- Clients which want to play Colormap tricks, such as colormap animation, need writable Colormap cells, and thus a dynamic Visual. For them, the class is the most important feature.
- Clients which require an exact color need a PseudoColor, DirectColor or TrueColor Visual. Other Visuals involve the server making an approximation to the requested RGB value. For these clients, the class is the most important feature.
- Clients that *really* care about the exact color will also be interested in the number of significant bits, which will tell them how closely the display can approximate the RGB value.
- Clients that require a certain number of colors to allow for contrast, but don't care about the exact colors, need a Visual with a Colormap large enough to hold the colors. For them, the Colormap size is the most important feature.

Xlib[1] provides two utility routines to make selecting a suitable Visual easier, using the *XVisualInfo* structure shown in Figure 3.

```
typedef struct {
        Visual          *visual;
        VisualID        visualid;
        int             screen;
        int             depth;
        int             class;
        unsigned long red_mask;
        unsigned long green_mask;
        unsigned long blue_mask;
        int             colormap_size;
        int             bits_per_rgb;
} XVisualInfo;
```

Figure 3: The XVisualInfo Structure

A client that simply wants to select a Visual of a given depth and class can use *XMatchVisualInfo()*. The client provides a depth and a Visual class, and an *XVisualInfo* structure describing one of the possibly many Visuals of that class and depth is returned.

A client that wants more detailed control over the selection of a Visual can use *XGetVisualInfo()*. The client constructs a template *XVisualInfo* structure, filling in the fields it is interested in. An array of *XVisualInfo* structures is returned, one for each Visual that matches the specified fields. For example, the code in Figure 4 will find all PseudoColor Visuals with 256-entry Colormaps.

```
{
        XVisualInfo     vinfo_template,
                        *vinfo_list;
        int             num_matching_visuals;

        vinfo_template.class = PseudoColor;
        vinfo_template.colormap_size = 256;

        vinfo_list = XGetVisualInfo(dpy,
                VisualClassMask | VisualColormapSizeMask,
                &vinfo_template, &num_matching_visuals);
        if (vinfo_list == (VisualInfo *) 0) {
                /* No such Visuals */
        } else {
                /* vinfo_list is an array of matches */
        }
}
```

Figure 4: Finding all 256-entry Colormap Visuals

In the case of the undump client, we have a problem that is more complex than either of these two simple cases. We have a set of RGB values that we wish to display as well as the available Visuals will let us. We have an upper bound N on the number of colors, but no other information about the colors‡.

Choosing a Visual capable of displaying N different RGB values takes four steps (the actual code is the routine *FindVisual()* in Appendix B):

- Rank the Visual classes in descending order of usability for our purposes:
  - TrueColor. A TrueColor Visual with large enough colormaps would be ideal for the dump client, since it would not merely represent the required RGB values, but it would not reserve private Colormap cells to do so.
  - DirectColor & PseudoColor. These Visuals, if their Colormaps are big enough, can represent the RGB values we need, but to do so they have to reserve private cells, reducing the server resources available for other clients.
  - StaticColor. A color Visual, even though we have no control over the color values, is likely to make a better approximation than a GrayScale Visual which cannot display colors at all.
  - GrayScale. This Visual, if its Colormaps were big enough, could make a monochrome approximation to the image better than a StaticGray Visual in which we would have no control over the shades of gray.
  - StaticGray. If there's nothing else, we'll have to make do with this.

‡ Scanning the RGB values could generate other information, such as that all the R, G and B values were equal in a gray image. Doing so would make the code, and the choice of a Visual too complex for a paper like this.

- Scan the list of Visuals returned by the server during the connection handshake, and for each Visual class record the Visual with the largest Colormaps. Of course, in the normal case, only a few of the classes will exist, and the others will be recorded as having a largest Colormap of size zero.
- In the rank order, look for a Visual with colormaps big enough to display the required number of colors. If one is available, use it.
- Otherwise, use the Visual with the largest Colormaps. Since none of the maps are big enough to display all the colors, the server will doing some approximation, and the bigger the map the better the approximation.

4.2. Converting RGB to pixel values

X provides three fundamentally different methods for converting RGB values to pixel values. A client can:

- Request the server to perform the conversion and return a pixel value. The result is to associate a sharable, read-only Colormap cell containing an approximation to the RGB values with the pixel value.
- Request the server to assign a writable Colormap cell and corresponding pixel value for the client's private use. The client can then set the Colormap cell to the RGB value.
- The client can predict the pixel value that will correspond to the RGB value, using its knowledge of the contents of the Colormap.

The choice of a suitable method depends on the requirements of the application and on the class of the Visual it is using.

4.2.1. Server does conversion

The AllocColor request takes an RGB triple and a Colormap as an argument, and returns a pixel index that points to a Colormap cell containing the best approximation the server can make to that RGB value, and the actual RGB values that the cell contains. It does so by:

- If the Visual is static (StaticGray, StaticColor, or TrueColor), it returns the cell in the map that is closest to the request RGB value.
- If the Visual is dynamic (GrayScale, PseudoColor, or DirectColor) and the Colormap is full, it returns the read-only cell with the closest RGB value to the requested one.
- If the Visual is dynamic, and the Colormap has free cells, the server will allocate an cell, mark it read-only, and set it to the requested RGB value.

This technique should be used by all applications except those for which the *exact* representation of a color is of primary importance, or which use very large numbers of colors. The reasons for preferring this method are that it will work on any Visual, and that it makes the most efficient use of the server's resources. Note, however, that since AllocColor returns a value, it requires a round-trip to the server and is therefore slow.

4.2.2. Client sets color

If the Visual is dynamic, clients can use AllocColorCells or AllocColorPlanes to reserve one or more pixel values for their private use. If the allocation succeeds, the pixel values will point to private, writable Colormap cells. The client can then use StoreColors to set the RGB values of these cells to exactly the values it desires.

This technique is more efficient than AllocColor, because a round-trip to the server is needed only for the initial allocation rather than every time a color is modified. However, it should be used only if it essential to the application because

- it will work only on a dynamic Visual,
- it consumes server resources that no other client can use until they are released,
- and unlike the other techniques, it can fail.

In general, clients using this technique should create a private Colormap. Ideally, the default Colormap for each Visual should be left for clients using the other techniques to maximize the sharing of resources.

Unfortunately, at present a default Colormap is available only for the Visual of the root window.*

4.2.3. Client predicts pixel value

There are two circumstances in which a client can determine the pixel value corresponding to a given RGB value *without* a round-trip to the server. To do so, it must have knowledge of the contents of the Colormap:

- If the Visual is TrueColor, the Colormap is known to provide linear ramps in each primary color. The R, G and B values can thus be adjusted to the match the corresponding sub-field mask, and or-ed together to make the pixel value.
- If the Colormap is one of the set of "standard" Colormaps a similar calculation can be performed. Applications wishing to use one of these Colormaps look for a property describing it on the root window; if the property is not found the application creates a suitable Colormap and a property to describe it.

Unfortunately, as specified the "standard colormaps" mechanism doesn't work if the server supports multiple Visuals. The properties describe a Colormap, but not the Visual it belongs to. Given a Colormap, the protocol does not provide any means to determine the Visual it belongs to. Thus, it must be assumed that the "standard" Colormaps belong to the default Visual.†

4.3. The Undump Algorithm

Here is an outline of the algorithm; the code itself is shown in Appendix B:

- Read in the width, height, number of colors & RGB values.
- Choose an appropriate Visual
- Create a Colormap for the chosen Visual.
- Create an image in memory to hold the pixel values that will eventually appear in the window.
- For each pixel in the data:
  - Convert the RGB values for the pixel into a pixel value, using the AllocColor request.
  - Store the pixel value into the corresponding location in the image.
- Create and map a window which uses the Colormap.
- Paint any exposed part of the window with the pixel values from the image.

4.4. Assessment of the Algorithm

Once again, this algorithm is about as simple as it can be and still get the job done. As a result, it is grossly inefficient. It makes the pessimistic assumption that every pixel in the image is a different color, and calls AllocColor to convert its RGB values into a pixel value. Of course, it is likely that many pixels are the same color, and calling AllocColor only for every distinct color would reduce the traffic considerably.

One problem is that this client creates its own Colormap. Ideally, clients should share Colormaps to improve the chance they will appear in their correct colors. If the "standard colormaps" mechanism worked for multiple Visuals, we could have used the RGB_BEST_MAP property to share a Colormap with other similar clients. If the default Colormap mechanism worked for multiple Visuals, we could have used the appropriate default Colormap.

The client will work on every Visual type, in most cases providing the best match available for the colors in the image. There is, however, one case in which the match will be less than optimal. If the Visual is PseudoColor and is too small to accommodate the number of colors in the image, the first colors encountered in the right-to-left within top-to-bottom scan of the image will get exact colors as new read-only cells are allocated by AllocColor. At some point, the map will fill up, and subsequent AllocColor calls will

* See section 2.2 of the Xlib manual. This problem is being addressed by the X Consortium.

† See section 9.2. of the Xlib manual. This problem is being addressed by the X Consortium.

return existing pre-allocated entries with approximations to the requested colors. The visual effect will be odd, as the quality of reproduction changes abruptly part way down the image.

The Visual selection process has two problems. For the TrueColor and DirectColor cases, it pessimistically assumes that all the distinct colors in the image differ only in one component, and will thus overestimate the size of the Colormap required. For example, suppose we have a blue image with 24 distinct colors; all 24 have the same Red and Green values, differing only in their Blue value. We would need a Colormap with 24 entries for Blue, but only 1-entry maps for Red and Green. In practice, it is likely that the color distribution will be more even and smaller maps would suffice.

Further, it is complicated by the fact that the number of distinct colors is only an upper bound. Consider a dump of a screen with an 8-bit PseudoColor Visual and a 1-bit StaticGray Visual. Assume that the image in the PseudoColor Visual had 256 different colors, two of which were white and black. The dump client would not realize that the white and black of the StaticGray Visual were the same as the white and black of the PseudoColor Visual and would record the number of colors as 258. Undumping to the same display, it would regard the PseudoColor Visual as inadequate to represent the 258 colors.

Better analysis of the set of RGB values to be displayed would solve both problems, but would make the program too complex for this paper.

5. Visuals and the Toolkit

The attention paid by the X Toolkit Intrinsics to the problem of specifying Visuals is evident from the fact that the word Visual does not appear in the index to the Intrinsics manual.[2] By reading the code, it is possible to determine the following information.

The Window for a Widget is created during the process of *realizing* the Widget, and the Visual to use has to be selected as part of this process. Different Widget classes will differ as to how they choose a Visual. All Athena Widgets use CopyFromParent as their Visual selection, deferring the choice to their parent. The root of the Widget tree is a so-called Shell Widget, which in the Intrinsics implementation itself inherits its Visual from its parent, in this case the root. So, all existing Widgets use the Visual of the root.

The core information, which all Widgets possess, includes a Colormap. For the Athena Widgets, this had better be a Colormap from the root's Visual, or Xlib errors will occur. This Colormap is used to convert RGB values to pixels for the Widget, using the AllocColor technique. One might think that it would be possible to realize a Widget in a Visual other than the root's by giving it a Colormap from that Visual, but this will not work.

A Widget class that needed some Visual other than the root's would have to implement its own realize procedure, and make the choice there. The choice algorithm would be the same for all instances of the Widget, but this does not mean that all instances would have to have the same Visual. This sounds onerous, but it is in fact the correct approach. As we showed above, the algorithm for choosing an appropriate Visual is application-specific; it would not be possible to wire-in to the Intrinsics a single algorithm that would satisfy everyone.

6. Conclusion

The X11 color model provides powerful access to the capabilities of different types of display hardware. Used carelessly, this access can lead to programs which run only in the environment in which they were developed.

All but the simplest X11 clients must take care to choose the most suitable Visual for their purposes, and to adapt their behavior to its capabilities. We have presented the techniques for doing so for a simple application, but more complex applications will require a wider range of techniques. More work is needed to develop these.

Acknowledgements

Thanks are due to James Gosling, who helped to develop our understanding of these issues, and who designed the multi-Visual and Colormap capabilities of the X11/NeWS server, to Jeff Vroom, who spotted the problem of the default colormap, to Bob Scheifler, who clarified several obscure points, and to the Usenix reviewers.

References

1. Jim Gettys, Ron Newman, and Robert W. Scheifler, *Xlib – C Language X Interface*, Massachusetts Institute of Technology, Cambridge, MA, 1988.
2. Joel McCormack, Paul Asente, and Ralph R. Swick, *X Toolkit Intrinsics – C Language X Interface*, Massachusetts Institute of Technology, Cambridge, MA, 1988.
3. David S. H. Rosenthal, *X Window System, Version 11: Inter-Client Communication Conventions Manual*, Massachusetts Institute of Technology, Cambridge, MA, 1988.
4. David S. H. Rosenthal, *A Simple X11 Client Program*, USENIX conference, Dallas, TX, February 1988. reprinted as *Going for Baroque*, Unix Review 6(6), June 1988.
5. Robin Schaufler, *X11/NeWS Design Overview*, pp. 23-35, USENIX conference, San Francisco, CA, June 1988.
6. Robert W. Scheifler and Jim Gettys, "The X Window System," *ACM Transactions on Graphics*, vol. 5, no. 2, pp. 79-109, April 1987.
7. Robert W. Scheifler, *X Window System Protocol, Version 11, Release 3*, Massachusetts Institute of Technology, Cambridge MA, 1988.

Appendix A: Dump Program

```
#include <X11/Xlib.h>
#include <stdio.h>
#define      max(a,b)      ((a) > (b) ? (a) : (b))
#define      min(a,b)      ((a) < (b) ? (a) : (b))

Display      *dpy;         /* X server we're talking to */
int dumpw, dumph;   /* Size of the "specified rectangle" */

/*
 *  There will be one of these for each Colormap we find
 */
struct cmp {
    struct cmp            *next;        /* Link in chain */
    Colormap        cmap;         /* The Colormap ID */
    unsigned long   allocated, used;/* Size & usage of inUse */
    XColor          *inUse;             /* Array of in-use colors */
};
struct cmp *inUse;  /* The list of Colormaps we found */
int numColor = 0;   /* Upper bound on number of distinct RGB values */

/*
 *  There will be one of these for each pixel in the rectangle
 */
struct pxl {
    struct cmp            *pCmap;             /* The colormap for this pixel */
    unsigned long   pixel;        /* The pixel value at this pixel */
};
struct pxl *Map;    /* The map representing the rectangle */
#define FindPixel(x,y) ((Map+((y)*dumpw))+(x))

/*
 *  Find (and create if necessary) a struct cmp for this Colormap
 */
struct cmp *
FindColormap(cmap)
    Colormap cmap;
{
    register struct cmp *pCmap;

    /*  If we've seen this Colormap before,  return its struct cmp */
    for (pCmap = inUse;  pCmap; pCmap = pCmap->next)
       if (cmap == pCmap->cmap)
            return (pCmap);
    /* First time, so create a new struct cmp, link it and return it */
    pCmap = (struct cmp *) calloc(sizeof (struct cmp), 1);
    pCmap->next = inUse;
    pCmap->cmap  = cmap;
    inUse = pCmap;
    return (pCmap);
}

/*
 *  Record this pixel value as being in use in its Colormap
 */
void
RegisterPixel(pixel, pCmap)
    unsigned long   pixel;
    struct cmp            *pCmap;
{
    register unsigned long i = pCmap->used;

    /*  If the pixel value is already known,  do nothing */
    while (i)
       if (pixel == pCmap->inUse[--i].pixel)
            return;
    /*  This is the first time we've seen this pixel value */
```

```
    if (pCmap->used >= pCmap->allocated) {
       /*  Need to expand or create the inUse array */
       pCmap->allocated = (pCmap->allocated * 2) + 10;
       pCmap->inUse = (XColor *)(pCmap->inUse ?
                    realloc(pCmap->inUse,
                          pCmap->allocated * sizeof (XColor)) :
                    malloc(pCmap->allocated * sizeof (XColor)));
    }
    /*  Now we have space to store the XColor, use QueryColor to get RGB */
    pCmap->inUse[pCmap->used].pixel = pixel;
    XQueryColor(dpy, pCmap->cmap, &pCmap->inUse[pCmap->used]);
    numColor++;
    pCmap->used++;
}

/*
 *  This gets called once for each window we find as we walk down the tree
 */
DoWindow(w, xo, yo, x, y, wi, hi)
    Window    w;
    int             xo, yo;       /* Parent's origin in root space */
    int             x, y;  /* Top-left of rectangle in root space */
    int             wi, hi;       /* Size of rectangle */
{
    XWindowAttributes      xwa;   /* Place to return the window's attributes */
    XImage          *xim;  /* Image to store window's pixels */
    int                    width, height, xl, yl, xi, yi;
    Window          root, parent, *children;
    int                    nchild, n;
    struct cmp             *pCmap;

    /*  Get the attributes of this window,  and locate its struct cmp */
    if (!XGetWindowAttributes(dpy, w, &xwa) || xwa.map_state != IsViewable)
       return;
    pCmap = FindColormap(xwa.colormap);
    /* Compute top-left of image in root space */
    xl = max(x, xwa.x+xo);
    yl = max(y, xwa.y+yo);
    width = min(x+wi, xwa.x + xwa.width + 2*xwa.border_width + xo) - xl;
    height = min(y+hi, xwa.y + xwa.height + 2*xwa.border_width + yo) - yl;
    if (width <= 0 || height <= 0)
       return;
    /*  Use GetImage to get the pixel values for the rectangle */
    if (!(xim = XGetImage(dpy, w, xl-xwa.border_width-xwa.x-xo,
                     yl-xwa.border_width-xwa.y-yo, width, height,
                     (~0), ZPixmap)))
       return;
    /*  For each pixel in the returned image */
    for (yi = 0; yi < height; yi++)
       for (xi = 0; xi < width; xi++) {
           register struct pxl *pPxl = FindPixel(xi+xl, yi+yl);

           /*  Label the pixel in the map  with this window's Colormap */
           pPxl->pCmap = pCmap;
           /*  And with its pixel value */
           pPxl->pixel = XGetPixel(xim, xi, yi);
           RegisterPixel(pPxl->pixel, pCmap);
       }
    /*  Free the space for the image */
    XDestroyImage(xim);
    /*  Find the children of this window, in back-to-front order */
    if (XQueryTree(dpy, w, &root, &parent, &children, &nchild)) {
       for (n = 0; n < nchild; n++) {
           /*  Process each of the child windows recursively */
           DoWindow(children[n], xo + xwa.x + xwa.border_width,
                   yo + xwa.y + xwa.border_width, xl, yl, width, height);
       }
       /*  Free the list of children */
       if (nchild > 0)
           XFree(children);
    }
    return;
}

/*
 *  Return the XColor structure for the pixel at [x,y] in the map
 */
XColor *
FindColor(x, y)
    int             x, y;
{
    struct pxl      *pPxl = FindPixel(x, y);   /* Find the struct pxl */
    struct cmp      *pCmp = pPxl->pCmap;              /* And the struct cmp */
    int             i;

    /*  Scan the in-use array for this colormap for the pixel value */
    for (i = 0; i < pCmp->used; i++)
       if (pPxl->pixel == pCmp->inUse[i].pixel)
           return (&(pCmp->inUse[i]));
    return (NULL);
}

/*
 *  Write the representation of the rectangle to stdout
 */
DoOutput(x, y, w, h)
    int             x, y, w, h;
{
    int             xi, yi;

    /*  Write the width,  height,  and number of colors */
    fwrite(&w, sizeof (int), 1, stdout);
    fwrite(&h, sizeof (int), 1, stdout);
    fwrite(&numColor, sizeof (int), 1, stdout);
    /*  For each pixel in the image */
    for (yi = 0; yi < h; yi++)
       for (xi = 0; xi < w; xi++) {
           XColor    *color = FindColor(x + xi, y + yi);
```

```
            /*  Write the R, G & B values for this pixel */
            fwrite(&(color->red), sizeof (unsigned short), 1, stdout);
            fwrite(&(color->green), sizeof (unsigned short), 1, stdout);
            fwrite(&(color->blue), sizeof (unsigned short), 1, stdout);
        }
}

main(argc, argv)
    int             argc;
    char      **argv;
{
    int             scrn;

    /*  Try to connect to the server */
    if ((dpy = XOpenDisplay(NULL)) == (Display *) 0) {
       fprintf(stderr, "Can't open display0);
       exit(1);
    }
    /*  Dump the specified part of the default screen */
    scrn = DefaultScreen(dpy);
    if (argc > 1)
       dumpw = atoi(argv[1]);
    else
        dumpw = DisplayWidth(dpy, scrn);
    if (argc > 2)
       dumph = atoi(argv[2]);
    else
        dumph = DisplayHeight(dpy, scrn);
    /*  Create the map with one struct pxl per pixel */
    Map = (struct pxl *) calloc(sizeof (struct pxl), dumpw * dumph);
    /*  Grab the server so things don't change under our feet */
    XGrabServer(dpy);
    /*  Recursively build the map */
    DoWindow(RootWindow(dpy, scrn), 0, 0, 0, 0, dumpw, dumph);
    /*  Finished reading things from the server - let it go */
    XUngrabServer(dpy);
    /*  Write the RGB representation of the rectangle to stdout */
    DoOutput(0, 0, dumpw, dumph);
    exit(0);
}
```

Appendix B: Undump Program

```
#include      <X11/Xlib.h>
#include      <X11/Xutil.h>
#include      <stdio.h>
#include      <sys/file.h>

Display                     *dpy;          /*  X server we're talking to */
Window              win;            /*  Window to paint the image in */
GC                  gc;             /*  GC to use for painting */
Visual              *visual;        /*  Visual to use for the window */
Colormap            cmap;           /*  Colormap to use for painting */
XEvent              ev;             /*  Event received from the server */
XImage              *image;             /*  To hold the image to be painted */
int                 width, height;      /*  Size of the window */
int                  best_size[6];      /*  Largest colormap per Visual class */
XVisualInfo             *best_vis[6];   /*  Best Visual per Visual class */
int                     best_class[] = { TrueColor, DirectColor, PseudoColor,
                                         StaticColor, GrayScale, StaticGray };

/*
 *  Return the most suitable Visual for representing numcolors colors
 */
XVisualInfo *
FindVisual(numcolors)
    int             numcolors;
{
    int                     num_vis, i, big_map = 0;
    XVisualInfo     vinfo_template, *vlist, *v, *big_vis = NULL;

    /*  Get descriptions of all the visuals */
    vlist = XGetVisualInfo(dpy, VisualNoMask, &vinfo_template, &num_vis);
    /*  Find the biggest colormaps for each visual class */
    for (v = vlist; v < vlist + num_vis; v++) {
       if (v->colormap_size > big_map) {
           big_map = v->colormap_size;
           big_vis = v;
       }
        if (v->colormap_size > best_size[v->class]) {
           /*  Biggest Colormaps seen so far */
            best_size[v->class] = v->colormap_size;
            best_vis[v->class] = v;
       }
    }
    /*  In decreasing order of usability,  look at each class */
    for (i = 0; i < 6; i++)
        if (best_size[best_class[i]] > numcolors)
            /*  This class can represent enough colors */
            return (best_vis[best_class[i]]);
    /*  Sigh!  We'll have to make do with a Visual that's too small */
    return (big_vis);
}

main(argc, argv)
int    argc;
char   **argv;
{
    int         numcolors, num_pxls, num_vis, x, y;
    unsigned short *rgbvalues;
    XVisualInfo *vis;
    XSetWindowAttributes values;
    XColor      color;
```

```
    /* Read the size information from stdin */
    fread((char *) &width, sizeof(int), 1, stdin);
    fread((char *) &height, sizeof(int), 1, stdin);
    fread((char *) &numcolors, sizeof(int), 1, stdin);
    /* Allocate space to hold the RGB data,  and read it in */
    num_pxls = width * height;
    rgbvalues = (unsigned short *) malloc(sizeof(unsigned short)
                                    * 3 * num_pxls);
    fread((char *) rgbvalues, sizeof(unsigned short), 3 * num_pxls, stdin);
    /* Connect to the server */
    if ((dpy = XOpenDisplay(NULL)) == (Display *) 0) {
       fprintf(stderr, "can't open display0);
       exit(1);
    }
    /* Find a suitable Visual for numcolors */
    vis = FindVisual(numcolors);
    /* Create a Colormap in the Visual we found */
    cmap = XCreateColormap(dpy, RootWindow(dpy, DefaultScreen(dpy)),
                    vis->visual, AllocNone);
    /* Create an image the right size */
    image = XCreateImage(dpy, vis->visual, vis->depth, ZPixmap, 0,
            (unsigned long *) malloc(num_pxls * sizeof(unsigned long)),
                    width, height, 32, 0);
    /* For each pixel in the image */
    for (y = 0; y < height; y++)
       for (x = 0; x < width; x++) {
           /* Fill out the RGB fields of the XColor struct */
           color.red = *rgbvalues++;
           color.green = *rgbvalues++;
           color.blue = *rgbvalues++;
           color.flags = DoRed | DoGreen | DoBlue;
           /* Get the server to convert from RGB to pixel value */
           XAllocColor(dpy, cmap, &color);
           /* Put the pixel value into the image */
           (void) XPutPixel(image, x, y, color.pixel);
       }
    /* Create a suitable window using the Colormap, background White */
    values.colormap = cmap;
    /* get White from our colormap */
    XAllocNamedColor(dpy, cmap, "white", &color, &color);
    values.background_pixel = color.pixel;
    /* Listen for Expose,  Enter and Leave events */
    values.event_mask = ExposureMask | EnterWindowMask | LeaveWindowMask;
    win = XCreateWindow(dpy, RootWindow(dpy, DefaultScreen(dpy)),
                    0, 0, width, height, 0, vis->depth, InputOutput,
                  vis->visual, CWColormap | CWEventMask | CWBackPixel,
                    &values);
    /* Create a GC to use for repainting the window */
    gc = XCreateGC(dpy, win, 0, NULL);
    /* Map the window,  wait for the Expose events,  and paint */
    XMapWindow(dpy, win);
    while (True) {
       XExposeEvent *e;

       XNextEvent(dpy, &ev);
       switch (ev.type) {
           case Expose:
              e = (XExposeEvent *) & ev;
              /* Copy the image to the exposed part of the window */
              XPutImage(dpy, win, gc, image, e->x, e->y, e->x, e->y,
                    (e->x + e->width > width ? width - e->x : e->width),
                    (e->y + e->height > height ? height - e->y : e->height));
              break;
           case EnterNotify:
              /* Mouse is in the window,  install its Colormap */
              XInstallColormap(dpy, cmap);
              break;
           case LeaveNotify:
              /* Mouse has left the window,  uninstall its Colormap */
              XUninstallColormap(dpy, cmap);
              break;
       }
    }
}
```

PEX – A 3-D Extension to X Windows

Spencer W. Thomas
EECS Department
University of Michigan

Martin Friedmann
Center for Information Technology Integration
University of Michigan

spencer@crim.eecs.umich.edu
martin@citi.umich.edu

December 1, 1988

Abstract

PEX (PHIGS Extension to X) is a proposed 3-D extension to Version 11 of the X Windowing System. It is intended to support the PHIGS and PHIGS+ graphics standards. During the summer of 1988, a demonstration implementation of PEX was produced and shown at the SIGGRAPH '88 vendor exhibition. The implementation is described, illustrating solutions to some problems induced by interoperability and portability requirements, and by the extreme potential variability in the data presented by the client program. The demonstration was successful, and the results are available on the X V11R3 release tape.

1 PEX

PEX[6] (PHIGS Extension to X) is a proposed extension to X11[8,5] providing three-dimensional graphics capability. It is intended to support the PHIGS[3] (Programmers Hierarchical Interactive Graphics System) graphics standard and the proposed PHIGS+[12] standard. PHIGS provides basic three dimensional graphics and modeling capability, PHIGS+ extends PHIGS to include shaded raster graphics and a few additional graphics primitives. The PEX protocol specification was first made public in early 1988[7,10], and was included on the X V11R2 release tape.

To support 3-D graphics, two new sets of requests and data structures are required. One set, clearly, consists of those that implement new graphics primitives, such as 3-D polygons (called "Fill Areas" in PHIGS terminology) or spline surfaces (NURBS). The second category contains the requests and structures that maintain the 3-D graphics state. Two major new structures are introduced in this second category: the renderer and the pipeline context. The renderer is the PEX analog of the X graphics context (GC). One must specify a renderer in order to draw PEX graphics. In fact, the renderer is bound to the drawable at the beginning of a sequence of rendering requests, and is not unbound until the sequence has completed. The reason for this tight binding is that there are some aspects of 3-D rendering (in particular, hidden-surface elimination) that cannot necessarily be determined until the data for a complete scene has been sent to the server.

The server also maintains a number of tables for color specification, point of view, light positions, and so on (these are required to support PHIGS and PHIGS+).

The PEX specification provides two ways for graphics to be drawn. One is an "immediate mode", in which primitives are drawn as they are sent to the server. The second mode supported by PEX is to have a server "structure store" (PHIGS terminology for a display list). Graphics primitives are loaded into the structure store, and then a "render structure" request is sent to the server, which renders all the data in the store. This is the only way in which PHIGS does graphics; the immediate mode is provided to allow for a client structure store, instead of keeping the structure store in the server.

2 The Demonstration Implementation

The demonstration implementation had two, sometimes contradictory, purposes. The primary purpose was to support a demonstration of PEX at SIGGRAPH '88. The secondary purpose was to provide a base for development of a full PEX sample implementation. When the two purposes came into conflict, the first was generally the winner. In addition, the demo was targeted at several specific hardware platforms, with the goal of supporting these as efficiently as possible, rather than providing easily portable generic display support (but see section 2.2.6 below).

Several explicit constraints were put on the design. The most important of these were

- No server structure store. This means that (as in the case of standard X windows) all data must be resent each time a window is updated.
- Only a few primitives supported: Polyline 3D, Fill Area 3D with Data, Annotation Text 3D, Quadrilateral Mesh, NURB Curve and NURB Surface.
- No support for multiple color formats.

In the interests of timely completion, and after examining the needs of the client programs, some additional constraints were decided on, including

- Static color table. The client can only select from one of a few colors. Thus, transmittal of "direct" colors from the client was unnecessary, and only indexed colors were supported.
- Static light table. It is possible to select which lights will be used, but their positions may not be changed.
- No per vertex or per facet colors. All primitives used the line, surface, and text color indices specified in the pipeline context.
- No change operations were implemented, all changes to the renderer or pipeline context were via output commands. Table entries (except for the view table) could not be changed.
- A simple color approximation method was implemented, allowing only shading of single colors (no interpolation between colors).
- The device-dependent implementations supported only a single device per server, and assumed that that device was present.
- In several places, it is assumed that the PEX display is screen 0.

2.1 Potential problems

Even before implementation began, there were some problems that would clearly need solution. They all had to do, in one way or another, with variability, and how to handle it efficiently. For example, data may be sent to the PEX extension in either of two floating point formats (VAX "F" or IEEE single precision). The only constraint is that all floating point numbers in a request must be in the same format. Colors, too, may be specified in one of 6 different formats, not all of which are the same size (they vary from 32 to 96 bits).

Finally, the request packets themselves contain a high degree of variability. As an example, consider the "Fill Area 3D with Data." This request represents a polygon with various data optionally associated with the polygon itself ("facet") and with its vertices. Specifically, the facet may be given a color and/or a normal vector. In addition, the vertices may each be given a color and/or a normal vector. The presence or absence of these data items is signalled by four bits near the beginning of the request.

One difficulty caused by all this variability is just that of accessing the data within the request packet. For transmission efficiency, only those data that are present are transmitted. Thus, the packet structure cannot be represented by a simple C structure. This may not seem so bad – after all, it is just an I/O problem, but in fact, it is desirable to keep the variability of structure, in order to save memory in the server. The solutions adopted in this implementation to these data variability problems are detailed below. In summary, they are: floating point is converted on input, color is not handled, and a stride-offset array structure is used.

Aside from the data structure representation issue, is the fact that the presence or absence of certain data items can have a significant effect on how a particular request is executed. Consider again the Fill Area example. It will be drawn quite differently if per-vertex normals or colors are present than if they are not. The problem is not so much in implementing these differences, but in doing so efficiently. The solution adopted in this implementation uses a combination of "if" statements with a procedure vector.

The third main source of variability is that not all graphics displays are the same. In fact, that should probably say "all graphics displays are not the same." This variability was handled through the use of a procedure vector for device-dependent drawing functions.

2.2 Implementation strategy

The basic architecture attempts to follow the structure of the core X server. There is a top level split between device independent functionality and device dependent functionality. The device dependent functionality is further subdivided by platform, and includes a "machine independent" (mi) device dependent implementation that uses the core X functions for drawing "through the GC."

Any tables used by the PEX extension are defined statically (with the exception of the view table.) None of the table manipulation requests are implemented. The server provides a predetermined set of colors (black, white, red, green, blue, yellow, cyan, magenta) to clients. The number of colors is limited so that a reasonable number of shades of each color can be provided on an 8-bit frame buffer.

2.2.1 Color approximation

A very simple color approximation method is used for the 8-bit displays. The color table is statically allocated, with a few color indices available to a PEX client. Each of these colors has several shades defined in the default X colormap at startup time. An assumption is made that

		⋮		
	0	0	0	← Base of red colors
Red shade 1 →	32	0	0	
		⋮		
Red shade 8 →	255	0	0	← True red color
	255	100	100	← Begin blending to white
Red shade 10 →	255	200	200	← Last red entry
		⋮		

Figure 1: Color map allocation

Allocation of color map entries for shades of red, with 9 shades from black to red, and two shades with white blended in.

these shades can be allocated contiguously in the X colormap. This would always be true if PEX were to allocate its own colormap, but this would mean that all other windows would be strangely colored whenever a PEX client was running. Since the demo PEX never releases its (shareable) colormap entries, there are only a few left for other applications.

The basic idea behind the color approximation scheme is that the only way in which a color will need to vary is via shading calculations. The colormap entries interpolate from black to the color, with a few more entries from the color towards white (for specular highlights). The allocation of colormap entries is optimized so that there is only a single black entry.

On a black and white display, a slightly different scheme is used. Instead of selecting an entry in the colormap, the shade is used to select a tile pattern. While the resulting shading is rough, it is much better than no shading at all.

Objects that are not shaded (polylines, text) are always drawn using the "true" color. If an object is shaded, the shade, as an integer from 0 to the number of shades, is used to index into the colormap from the base of the range assigned to the given PEX color to get the color index to use to draw the object. This is illustrated in Figure 1.

There are two major reasons for using this scheme. The first is that it is simple and fast. The second is that on displays that will do hardware color interpolation, this is about the only way to be able to use the hardware for "Gouraud" shading. The main disadvantages of the method are that only a limited number of colors are available, and that the output looks "banded." Given that the alternatives are very slow shading or the use of specialized hardware, we feel that it is a good tradeoff.

2.2.2 Overall architecture

This section will discuss the basic architecture of the system, including the division of the code into device independent and device dependent sections; some interesting data structures and their implementation; and data type (floating point) conversion (see also [10]).

The core X server is divided into device independent ("dix") and device dependent ("ddx") routines[1,2,9]. The interface between the two is, with some exceptions, carried as a procedure vector in a "graphics context" (GC) data structure. In addition to the procedure vector, the

GC carries graphics state information, such as fill mode (e.g., solid or tiled), foreground and background pixel values, tile pattern, etc. All output operations specify a GC and a drawable (window or pixmap). Prior to actually drawing, the GC and drawable combination is "validated." Besides performing necessary consistency checks, validation loads into the procedure vector the appropriate routines for drawing in the specified mode into the given drawable.

This provides an efficient way to create and use arbitrarily specific drawing functions. For instance, if it there was a very efficient way to tile fill a rectangle, provided that the window lay entirely in the upper right quadrant of the screen and the tile pattern was a certain size, a function could be written that handled only that case. This function would be placed into the "fill rectangle" procedure slot only when the conditions for its use were satisfied, while otherwise a more general fill routine would be used. Presumably the state of the GC would be changed less frequently than once per filled rectangle, so the cost of determining the special condition would be spread over several function invocations.

Perhaps more importantly, the GC procedure vector allows the determination of the function to be called for any output operation to be made solely by the device dependent code layer. Thus, the device independent part of the X server can remain unchanged over the range of displays it can potentially drive.

A "ddpex" interface was defined to provide an interface between device independent rendering code and the hardware capabilities[11]. Since the 3-D rendering capabilities of workstations varies so much from one to the next, a layered approach, with three layers, has been taken. It is not necessary for a device interface to provide functions at all layers, if all the required functionality is available "in hardware." It is expected that this approach will not lead to inefficiencies, since the layering is coarse, a fair amount of work is done at each layer, and the overhead of an "extra" function call will be minimal.

The three layers of the interface correspond to, in increasing order of complexity, the pixel level, a 2-D rendering level, and a 3-D rendering level, including transformation support. The pixel level would be used for those functions that are not supported at all by the hardware (e.g., shading with normal interpolation), in which the complete rendering pipeline is implemented in software.

The 2-D rendering level is used for drawing primitives that have been completely transformed and clipped. Clipping prior to calling the 2-D layer is necessary to properly accomplish Z clipping and perspective division.

The 3-D rendering level is used for those devices that support a complete 3-D rendering pipeline, including transformation and clipping. For functions that are not supported in the hardware (e.g., shading with normal interpolation) it would be necessary to bypass the 3-D layer and use the software rendering pipeline. It may be possible to use the device transformation (and clipping) pipeline before passing the data on to the 2-D layer.

In a full PEX implementation, device independent implementations would be provided for all functions at the 2-D and 3-D layer. Device dependent implementations must be supplied for at least the pixel layer if no higher level functions are provided. However, if all functionality is available via the device dependent 3-D layer functions, the 2-D and pixel layers do not need to be provided.

Ddpex functions are invoked via a procedure vector in the Renderer. The procedure vector is set up by calling a validation procedure initially (at Begin Rendering time, for example), and whenever the renderer or pipeline context is modified (analogously to the case of GC validation). Depending on the renderer and pipeline context mode values, different procedures may be placed in the vector slots.

The demonstration implementation used only the 3-D layer functions, as most of the targeted

```
#define Pex1DData(pObj, n, data)\
    ((char *)(pObj)+\
     ((n) * ((pObj)->data##Stride))+\
     ((pObj)->data##Offset))
...
typedef struct _pexFillArea3DwithData
{
...
    pexBitmaskShort      vertexAttributes;
...
    CARD32               vertexNormStride;
    CARD32               vertexNormOffset;
    CARD32               vertexStride;
    CARD32               vertexOffset;
...
    /* Variable Sized Portion is appended here */
} *pexFillAreaWithDataPtr, pexFillAreaWithDataRec;

#define PexFillAreaVertex(o, n)\
    (pexCoord3D *)  Pex1DData(o, n, vertex)
#define PexFillAreaVertexNorm(o, n)\
    (pexVector3D *)  Pex1DData(o, n, vertexNorm)
```

Figure 2: Example of variable-sized array use

hardware platforms had 3-D graphics capability. The "machine independent" (mipex) ddpex implementation (see below) could probably have benefited from availability of 3-D to 2-D layer device independent functions, but this was not done (mostly because of time constraints).

A few device independent rendering functions were provided. With the exception of the NURBS code, these are simple utility functions for initializing the static lookup tables, clipping lines, and transforming points. The NURBS code provides 3-D ddpex functions for spline rendering that translate curves into polylines, and surfaces into quadrilateral meshes, and then call the appropriate 3-D ddpex function to complete the rendering[10].

2.2.3 Variable size arrays

One interesting feature of the data structures used is the implementation technique for variable sized arrays of variable sized elements. Each repeated data element (such as a vertex, vertex normal, or facet normal) is accessed via an offset (to the first element of the list) and a stride (number of bytes between two consecutive list elements). The stride and offset values are set at the time the data structure is created.

A set of macros is provided to simplify accesses. For example, a Fill Area 3D with Data contains a LIST of VERTEX. A VERTEX is an OPT_DATA and a COORD3D. An OPT_DATA may contain a normal vector (COORD3D) and a color. The relevant macros and portions of data structure are shown in Figure 2[1].

The `Pex1DData` macro is a general purpose access function for lists. There is a corresponding macro for accessing two dimensional arrays (as found in quadrilateral meshes and NURB

[1]Note that ANSI C token concatenation is used in this example. The actual code uses an "ifdef" to determine the type of token concatenation to use.

surfaces). The desired datum is located by adding the offset to the object address (taken as a char pointer), and then adding the list index times the stride. Assuming a good compiler, this should be as efficient as regular array subscripting. In the fill area, the presence or absence of vertex normals is determined by the attribute bit mask.

Any area of memory can be treated as a list or array of arbitrarily sized elements by using the `pexArray1` and `pexArray2` structures. These are simple array descriptors, containing a pointer to the body of the list or array; the number of elements in the list, or the number of rows and columns in the array; and a stride, or a row stride and a column stride. The macros `PexArray1Index` and `PexArray2Index` provide random access to list or array elements. For sequential access, `PexArray1Next` returns a pointer to the next element in a list, given a `PexArray1` descriptor and a pointer to the current element. `PexArray2NextCol` and `PexArray2NextRow` perform similar functions for two dimensional arrays.

2.2.4 Control flow

The basic control flow is quite simple. One of the two dispatch routines (for swapped or unswapped data) is called by the core server. The dispatcher checks to see if the request contains floating point values. If not, the appropriate request handler is called through a procedure vector (there are separate vectors for swapped packets and unswapped packets). If the request contains floating point values, it is further checked to see whether the floating point format is the same as the server's native format. One of four procedures (no swap and no float conversion, swap and no float conversion, no swap but convert floats, and swap and convert floats) is called through the appropriate procedure vector. The return code from the handler is passed back to the server as a status value.

2.2.5 Data conversion

For efficiency, any function that must deal with "raw" request data (that is, data that still has the client byte order and floating-point format) exists in two or four incarnations (two if the request has no floating point numbers, four otherwise). To avoid programmer tedium and parallel maintenance headaches, the code was written using macros and (if necessary) conditional compilation, so that a single source could produce all the necessary versions. For example, the macro `SWAPFLOAT` might do nothing, just swap bytes, just convert floating point format, or swap bytes and convert floating point format, depending on how it was defined. If the symbol `NO_CONVERSIONS` was defined to be 1, then a simple assignment or bcopy could be conditionally compiled when request data must be copied. The value of `FUNCTION_PREFIX` is a token that is concatenated to the beginning of the function name to distinguish the four incarnations.

Four include files (`NoConv.ci`, `OnlySwap.ci`, `OnlyConv.ci`, and `SwapConv.ci`) set these macros to appropriate values. The `ci` extension indicates that these files contain neither "header" information nor C program text, but "included" C code. The source for the functions to be replicated is also found in files with the `ci` extension. Finally, the two files `sgen.ci` and `scgen.ci` cause the generation of, respectively, the two or four routines from the template. A code fragment that generates the various UnpackPexPolyline functions, may be illustrative.

```
/* polyline.c */
...
#define SWAP_FILE "polyline.ci"
#include "scgen.ci"
#undef SWAP_FILE

/* polyline.ci */
...
pexPolylinePtr
FUNCTION_PREFIX##UnpackPexPolyline(poly)
    pexPolyline *poly;
{
...
#if NO_CONVERSIONS
        bcopy((char *)poly + sizeof(pexPolyline),
              (char *)ReturnPoly + sizeof(pexPolylineRec),
              (int)(nPoints * sizeof(pexCoord3D)));
#else
        destPtr = (FLOAT *)(((char *)ReturnPoly) +
sizeof(pexPolylineRec));
        maxPtr = (FLOAT *)(((char *)poly) + sizeof(pexPolyline) +
                           (int)(nPoints * sizeof(pexCoord3D)));

        for(fromPtr = (FLOAT *)(((char *)poly) + sizeof(pexPolyline));
            fromPtr < maxPtr;
            fromPtr++, destPtr++ )
        {
            /* Beware: swapfloat expands to multiple statements */
            pSWAPFLOAT_C(fromPtr, destPtr);
        };
#endif /* no_conversions */
...
}
```

This makes UnpackPexPolyline, sUnpackPexPolyline, fUnpackPexPolyline, and sfUnpackPexPolyline.

2.2.6 Machine independent PEX driver

In order to use PEX on displays with no special 3-D graphics hardware, a "machine independent" (mipex) device dependent driver was written[4]. It uses the core X server's line, text and polygon drawing functions as its 2D and pixel drawing layer.

The primitive rendering functions are logically split into two parts. First, a simple rendering package is used to transform, clip and shade the primitive into device coordinates and color indices. Second, the core X functions for drawing through the GC are used to realize the primitive in the frame buffer. Since these functions use the GC for all actual drawing, they should work with little modification when linked with any X server provided that the server's architecture resembles the Sample Server provided by M.I.T.

One interesting feature of the mipex driver is how "smooth" (Gouraud) shading is handled. The shading method takes advantage of the small number of shades provided by the color approximation method. As implemented, it works only with triangles or quadrilaterals (this was sufficient for the demonstration).

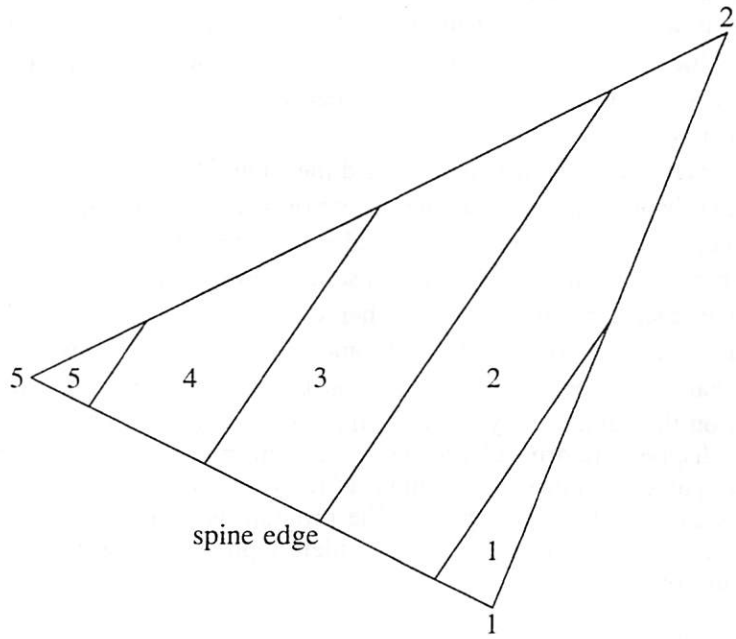


Figure 3: Triangle slicing for smooth shading

A triangle with shade values of 1, 2, and 5 is subdivided into 5 flat-shaded polygons with shade values of 1 through 5.

In fact, the smooth shading code only works with triangles, so quadrilaterals are first split (carefully) into two triangles. The algorithm hinges on the fact that the triangle can be split along its edges according to the color gradient, and drawn in several calls to the X FillPolygon function in the GC. It is a slice-and-dice approach to the problem that avoids having to set individual pixels in the frame buffer on machines that do not provide hardware smooth fill.

To clip the triangle each of the small polygon slices must be clipped separately, because if the whole triangle were clipped, it could come back with more than four vertices. To speed the process, the polygon is tested to see if it will be clipped at all. If not, none of its pieces are sent to be clipped. At the same time, a test is made to see if the whole polygon is clipped out. If so, the polygon is not sliced up.

A shade is calculated for each vertex. The edge with the maximum color gradient is identified and is called the spine edge. The sum of the gradients of the other two edges must equal that of the spine. The triangle is sliced linearly along the spine, with connecting points on those other two edges (see Figure 3).

Since the routine can handle 4-sided polygons, a quadrilateral mesh can be smooth shaded. And, since a NURB surface is drawn as a quadmesh, those can also be smooth shaded.

3 Results and conclusions

The demonstration shown at the SIGGRAPH '88 vendor exhibition consisted of two different applications. One was designed to show interactivity and interoperability between machines,

while the other illustrated the various rendering capabilities of the server. There were three "PEX" machines at the show, one in each of the DEC, HP, and Sun booths. An ethernet connected the machines. In addition, two "server" machines were available; these did not have displays, but were sometimes used for running the game (see below) so that it did not have an adverse impact on any of the displays.

The first application is a multi-player three-dimensional "pong" game. Up to six players can participate, one per display (at the show, there were only three displays, so this limited the number of players to three). Each player sees the cubical room from the point of view of one face (wall). The object is to move a "paddle" using the mouse to keep the ball from striking the wall. A point is given to a player each time the ball strikes her wall. The player with the lowest score "wins." The paddle floats slightly in front of the wall, and may be moved in two dimensions only. It is possible for the ball to get behind the paddle and bounce between it and the wall, leading to a second variation on the game: to try to get as high a score as possible.

All the game displays are driven by a single program, running on another machine (preferably not on one of the game machines, as it competes for CPU cycles with the server) that computes the ball positions and updates the displays. The program is written so that it does not send any updates to each server until that server has completely processed the previous batch of updates. The main program loop is:

```
check if any displays are ready
if so, for each one
    get the mouse position
    update the paddle position
update the ball position (check for wall, paddle collisions, etc.)
for each display that is ready,
    update the display
```

The game was successful, and a hit (although we did not discover a couple of crucial program bugs until after the show). On all displays, the update was quick and smooth enough to make the game playable. (It even retains some amount of playability on a black and white Sun 3/60.) It did, however, show the need for some sort of double buffering capability, as the displays without it tended to blink.

The second application is less dynamic. It is a "CAD review" application – the user selects from a menu of B-spline models of objects ranging from a soft drink can to a mechanical pencil. The object is displayed in its own window, and can be viewed from different positions and angles by manipulating a set of slider controls. Another menu is used to change the various graphics parameters, such as wire-frame versus shaded display, hidden surface elimination, smooth shading, and color. Several object display windows may be open simultaneously. While not providing "interactive" update rates for most models, the performance is acceptable, and is certainly better than that achieved by doing all the 3-D graphics on the client side, and using standard X calls to draw the result. This is due mostly to the reduced data transmission required when using PEX.

The source code for the demonstration PEX extension, a number of test programs, and the "pexpong" game are on the X V11R3 release tape, along with documentation for the implementation (including many of the references in this paper). It is, however, a demonstration implementation. There are still many holes in it, and probably a few bugs. We hope that others may find it useful, though.

We feel that the demonstration shows that PEX is practical as an X extension, and that reasonable (even good) performance can be achieved doing 3-D graphics "across a wire." Most of the lessons that we learned during the project are not unique to PEX, but are common to most

large software development projects. The most important lesson was not to agree to do four months worth of work in two months, even though you think you really want to.

4 Acknowledgements

The PEX demonstration was created under the auspices of the University of Michigan Center for Information Technology Integration (CITI), director Bertram Herzog, by Spencer W. Thomas, Martin Friedmann, Charles P. Jerian, Cheryl Huntington, Paul Martz, and David Whipple, with assistance (gratefully received) from Bert Herzog and Andy Goodrich. The demo could not have come off without the hard work of Dave Bachmann, Greg Buzzard, Kathy Kuisel, Deborah Swanberg, Alice Gordenker, Todd Newman, Marty Hess, and Jeff Stevenson. This work was supported by Digital Equipment Corporation, Sun Microsystems, and Hewlett-Packard Corporation. All opinions expressed herein are those of the authors, and do not necessarily reflect the views of the sponsors.

References

[1] Angebranndt, S., Drewry, R., Karlton, P., Newman, T., and Scheifler, B. *Definition of the Porting Layer for the X v11 Sample Server*. Massachusetts Institute of Technology. Included in X V11 distributions.

[2] Angebranndt, S., Drewry, R., Karlton, P., Newman, T., and Scheifler, B. *Strategies for Porting the X v11 Sample Server*. Massachusetts Institute of Technology. Included in X V11 distributions.

[3] ANSI. *Programmer's Hiearchical Interactive Graphics System (PHIGS) Functional Description*. Document number X3.144-198x.

[4] Friedmann, M. *The Machine Independent Rendering Interface*. CITI, University of Michigan. Included in X V11R3 distribution.

[5] Gettys, J., Newman, R., and Scheifler, R. W. *Xlib – C Language X Interface*. Massachusetts Institute of Technology. Included in X V11 distributions.

[6] Rost, R. Adding a dimension to x. *Unix Review 6*, 10.

[7] Rost, R. *PEX Introduction and Overview*. Included in X V11R2 distribution.

[8] Scheifler, R., and Gettys, J. The x window system. *ACM Transactions on Graphics 5*, 2 (April 1986).

[9] Susan Angebranndt, e. a. *X11 Server Extensions*. Massachusetts Institute of Technology. Included in X V11 distributions.

[10] Thomas, S. W. *Architecture of the PEX Demonstration Extension*. CITI, University of Michigan. Included in X V11R3 distribution.

[11] Thomas, S. W. *Preliminary DDPEX Description*. CITI, University of Michigan. Included in X V11R3 distribution.

[12] van Dam, A. Phigs+ functional description revision 3.0. *Computer Graphics 22*, 3 (July 1988).

XVT: A Virtual Toolkit for Portability Between Window Systems

Marc J. Rochkind

Advanced Programming Institute Ltd.
Boulder, CO

Abstract

The Extensible Virtual Toolkit (XVT) is a high-level interface that allows graphical, interactive applications to be easily ported to various window systems, such as X-11, MS-Windows, OS/2 Presentation Manager, and the Macintosh. Behind the common interface there is a separate implementation in the form of a C object library for each host system.

This paper describes the design principles behind XVT and its key programming features. It then reviews the main problems in creating an implementation for X and explains our short-term solutions. Also discussed are plans for more thorough long-term solutions using industry-standard toolkits.

PORTABILITY BETWEEN WINDOW SYSTEMS?

Software designers have struggled for portability since the earliest days of computing. Originally, starting in the late 1950s, portability concerns focused on the standardization of programming languages—Cobol and Fortran—across different machine architectures and operating systems. By the mid-1970s, with C and UNIX, portability had been extended to system programming languages and operating systems. The development in the mid-1980s of X and NeWS has proved that even window systems can be portable.

With the compiler, operating-system, and window-system implementors doing most of the work, application programmers lucky enough to restrict themselves to these standardized environments—a combination, say, of UNIX and X—have gotten a free ride. For them the portability issue is essentially solved. But computers running proprietary system software, mainly personal computers, still account for a huge market. Application developers who want to go after it still have to deal with portability across operating environments, only one of which is UNIX and X.

Fortunately, nearly all of today's computers have C compilers, so at least the language problem is tractable; the advent of ANSI C compilers will improve the situation even more. But portability across operating systems, and, for many developers, across window systems is still an obstacle, and it will remain one for a long time. The current generation of popular PCs, Macintoshes and IBM compatibles, will probably never run UNIX and X as their primary systems. And even if they do, it won't be for years.

So developers have three options: They can target UNIX and X systems, and ignore the rest; they can recode their programs for each environment; or they can write to a common interface that hides the differences between the underlying operating and window systems (much as the Standard I/O library for C hides differences between file systems).

Clearly, the third choice is the most attractive. But is portability across window systems possible? And, if it is, is it practical? Are the various window systems and user interface toolkits similar enough in their basic capabilities? Will portable applications be fast enough? Small enough? Will too many features have to be sacrificed? After all, history has shown that

portability becomes accepted not when it can be first be demonstrated, but when it first becomes reasonably competitive with non-portable approaches in terms of efficiency.

The Extensible Virtual Toolkit (XVT) is, as far as we know, the first commercially available C library for portability across window systems. It was first shipped for the Macintosh and MS-Windows in January, 1988, and there is now substantial evidence that such portability—at least across those environments—is indeed practical. We are now completing XVT libraries for OS/2 running Presentation Manager (PM), for X-11, and for a character-based window system [Roc88] (Fig. 1).

OS/2-PM is a successor to Windows, so that port presented no conceptual difficulties, just lots of tedious details to be changed. The X-11 job was tougher, and is the subject of most of the rest of this paper.

The character-based implementation is by necessity incomplete; XVT calls such as `draw_oval` aren't implemented at all, and others, such as `set_font`, are only partially implemented (you can set the text style to bold, for example, but you can't set the face name or point size). Yet, we were surprised at the number of XVT functions that made sense as character-oriented operations. Event handling, window creation and manipulation, menus, dialogs, and printing all have equivalents in the character-based world. They operate with a choice of coordinate systems: A 25-by-80 system in which the characters are numbered, and a 250-by-800 system in which the mathematical lines between the pixels are numbered and the characters are all 10-by-10 pixels in size. The pixel coordinate system makes porting to a graphical system easier, whereas character coordinates are more natural for those already used to character displays.

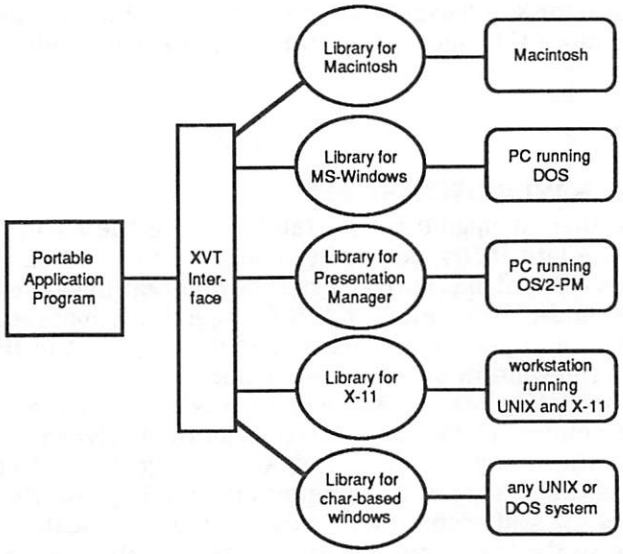


Figure 1. The XVT interface shields a portable application from details of a particular window and operating system. This picture was created with XVT-Draw, a portable drawing program written with XVT.

Ground Rules for the Design of XVT

In designing a portable window system interface it's important not to aim for too low or too high a level of abstraction. At the extreme low end you have C itself, which offers portability but hardly qualifies as a window system. At the high end you have application programs like PageMaker, which runs on several window systems. What we want is in between: Its function calls should look like those of a window system and toolkit (with events, windows, graphics, fonts, a clipboard, and so on), but it should be simpler than any of the underlying systems on which it runs.

So we established some ground rules to ensure that XVT would be at the right level of abstraction, would solve the right set of problems, and would support the right kinds of computers and window systems:

- It must be practical to program real applications, not just toys, and those applications must include highly graphical ones. This can be tested only by releasing the package commercially; a few examples coded by XVT's developers don't prove anything.
- At least X, MS-Windows, OS/2-PM, and the Macintosh must be supportable, and, perhaps eventually, NeWS. These environments are sufficiently dissimilar that abstractions designed to bridge them are likely to work for other systems, too.
- The XVT interface must be thin; that is, it must use whatever the host window system and toolkit offer for window management, menus, resources, dialogs, and so on. XVT must not be yet another window system that interfaces with the hardware or requires vast amounts of memory.
- XVT programs must be written in ordinary C, with no special pre-processing or interpretation. That is, XVT is just a library and some header files, not a new programming environment in its own right.
- On a given target machine, the application must look like it belongs there, with a look and feel that's compatible with native applications. This is crucial in the Mac community, less so in the others. But it is likely to become increasingly important even for X.
- No conditional compilation or execution that depends on the native toolkit should be required in application programs. Portability must be achieved through abstraction, not through multiple paths.
- Resources (precompiled definitions for menus and dialogs) don't have to be virtualized. They can be rewritten for each native system. (Portable resources are probably possible too, but this restriction reduced the labor to get the initial version of XVT completed.)
- The interface must be extensible, in that it must be possible for applications to access underlying window system features. This reduces portability, of course, but it ensures that application developers won't find themselves boxed in if they need a few features that aren't part of the XVT model. Surely 90% portability is better than none!

These ground rules aren't the only viable ones. With different rules other solutions to portability across window systems are possible, and under the right conditions each of them is optimal. For example, if stylistic issues and the need for graphics aren't important, then an ordinary classical UNIX-style program (with an "argc, argv" style user interface) is portable, since all of the target systems (even the Mac) support such programs. Without the requirements for C, for a thin interface, and for applications that look native, Smalltalk-80 is a good portability solution.

There isn't room in this paper to describe the XVT interface in detail. In the next section I'll try only to mention its key features and to indicate its scope. Then I'll show the obligatory example program that displays (what else?) "Hello World!". Finally, I'll discuss the main challenges we faced in implementing XVT for X. The remainder of this paper assumes general familiarity with X [Nye88].

A Quick Look at XVT

It's convenient to look at XVT, or any window system, in terms of four levels. The *operating system* level includes device support, networking, task management, memory management, and a file system. The *window* level includes windows, events, graphics, fonts, cursors and carets, and resources. The *user-interface toolkit* level includes support for menus, dialogs and alerts, window controls (e.g., scroll bars), printing, and the clipboard. Finally, the *style* level includes look-and-feel policies and desktop management.

For the PC window systems these four levels aren't clearly demarcated. The Mac is the most monolithic: Its operating system doesn't even have a name (some call it the "finder," which is roughly equivalent to calling UNIX the "Bourne shell"). MS-Windows and Presentation Manager do run on distinct operating systems, but neither they nor the Mac distinguish between the window and toolkit layers. These systems also support a particular look and feel; although de-

velopers can stray from it if they wish (there's more than enough flexibility), Apple and Microsoft strongly discourage it.

X is much more modularized. X itself handles only the window layer—it was never intended to do more. The operating system level is UNIX. The toolkit can be chosen from an ever-growing list (e.g., Xt, Sony Widgets, DEC Windows). A consistent style is optional: Each application can have its own, or a particular look-and-feel, such as Open Look, can be mandated.

Because XVT is designed to support real applications, which need support on all four levels, its abstractions have to encompass a wider scope than only the window layer. The PC operating systems are weak relative to UNIX, so, to ensure portability, XVT has to support memory allocation (to handle large blocks outside an application's local heap) and file operations, such as changing directories and reading files with line-ending sequences that include almost all permutations of carriage return and linefeed. Even ANSI C functions such as `ctime` have to be included in the XVT interface, because the current Mac compilers don't match the standard, or even the UNIX C library, very well.

At the window level XVT includes features to create and destroy windows; to change or query their position, size, title, activation, and clipping rectangle; to handle events; to draw graphics, with various pens, brushes, and transfer modes; to select fonts and measure text; to manipulate the cursor and caret (insertion point); and to access resources. The XVT feature list in this layer essentially parallels that of X, but X is somewhat richer and more complicated.

XVT includes user-interface toolkit features as first-class citizens, as the Mac, MS-Windows, and Presentation Manager do, rather than as an optional add-on, as X does. There are functions to access or modify menus; to put up modal or modeless dialogs and alerts, with push buttons, radio buttons, check boxes, editable and static text, list boxes, and scroll bars; to manage user-activity during dialog processing; to put text, pictures, and application-defined objects onto the clipboard, and to get them off; and to print.

At the style level XVT needs to do little, as most look-and-feel issues are up to the underlying window system. However, XVT directly supports standardized dialogs for choosing file names, and an elaborate online help system. Guidelines in the XVT programmer's manual help the designer code resources so as to follow the style of each host system while maintaining a uniform interface to the program, which must be invariant across systems.

Physically, XVT consists of a single interface, `xvt.h`, which is identical across all systems. It contains about 200 function prototypes along with type definitions for abstract objects such as windows and fonts. There's a separate implementation of this interface for each host system in the form of an object library. In the case of X, a client program is linked with the library `xvt.a`. Normally, a client makes no Xlib calls for itself—most of its XVT calls result in Xlib calls by XVT. XVT makes no use of any X internals, nor does it require any modification of X. Conceptually, we treat X as a black box (like the Mac ROM) and don't attempt to exploit the fact that we have access to its source code. We don't supply X for workstations, but rather run on the port provided by the hardware vendor.

The preceding paragraphs have only scratched the surface of XVT, just to make the point that XVT is broad, in that it spans all four levels, and deep, in that its feature set is reasonably complete. More details on XVT itself are in [Roc87].

HELLO WORLD IN XVT

It's traditional when writing about window systems to include a "Hello World!" program. The following short but entirely complete* XVT program is our competitive entry. (If anybody comes up with a shorter program than this for any window system, we're prepared to fight back by eliminating the feature that allows the user to quit.)

*One of the referees who reviewed a draft of this paper commented that we were cheating by including a program segment, rather than a full program. We plead innocent! The program shown is indeed complete—details that appear to be missing, such as building an initial window or specifying events to be processed, are instead handled entirely by XVT.

```
#include "xvt.h"
#include "xvtmenu.h"

void main_event(win, ep) /* called when an event occurs */
WINDOW win;              /* window affected by event */
EVENT *ep;               /* pointer to event structure */
{
     RCT rct;

     switch (ep->type) {
     case E_UPDATE:
          get_client_rect(win, &rct);
          set_pen(&white_pen);
          draw_rect(&rct);
          draw_text(10, 100, "Hello World!", -1);
          break;
     case E_COMMAND:
          if (ep->v.cmd.tag == M_FILE_QUIT)
               terminate();
     }
}
```

The appendix shows a more interesting program that allows the message to be changed (to "Goodbye World!") from the keyboard, with the mouse, or by choosing a menu command. The message appears in a custom-sized rounded-corner rectangle. The user can change font and text-style, and even open up multiple windows. Figure 2 shows this program running on a Mac. The same source can be compiled and run on an IBM-compatible running Windows or OS/2-Presentation Manager, or on a UNIX workstation running X.

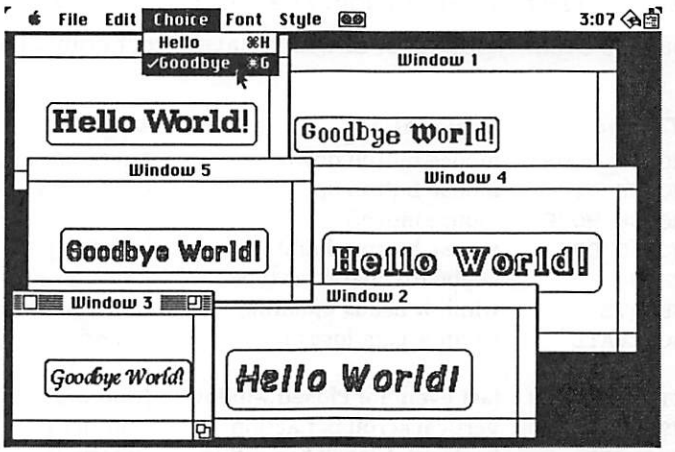


Figure 2. Six hellos and goodbyes from the XVT example program shown in the Appendix.

THE X IMPLEMENTATION OF XVT

This section discusses some of the more interesting problems we encountered, and are still encountering, in building an XVT implementation for X-11. We were pleased to discover that some features, such as graphics, moved over quite easily, and others (such as our implementation of `stat`) could be skipped altogether. We were not pleased that other features, such as printing, turned out to be major programming projects. We'll break this discussion into the levels outlined earlier: operating system, window, and user-interface toolkit. A few comments about the style level will be made as well.

Operating System Level

X is much richer in operating system features than are the PC window systems. More precisely, it's not X that has these features, but rather UNIX.

The PCs usually lack memory management hardware, and the Mac and DOS operating systems ignore it even when it is there, so they use a scheme involving doubly-indirect references to allow them to rearrange the heap without invalidating application-held addresses. In XVT the call `galloc` allocates a block and returns a handle; `glock` locks it and returns a pointer; `gunlock` frees the pointer (but retains the handle) so the memory manager can move the block if it needs to. Implementing these functions on UNIX is easy: `galloc` is a macro that calls `malloc`, `glock` is the identity function, and `gunlock` is a no-op. The only disadvantage of this implementation is that proper use of `glock` and `gunlock` can't be tested—an accidentally unlocked pointer would remain valid. (Programs that shouldn't work but do are even harder to test and debug than program that should work but don't!)

XVT's directory manipulation interface maps onto UNIX system calls easily. Its implementation was taken from the DOS version of XVT, since the UNIX and DOS file-system calls are similar.

Most of the other operating system features in XVT (such as `ctime`) aren't needed at all on UNIX.

Window Level

This is X's forte. We didn't anticipate any fundamental problems in implementing this part of XVT, and we found none. There were lots of niggling details to be addressed, however.

Window Manipulation—XVT's `new_window` and `close_window` map well into `XCreateWindow` and `XDestroyWindow`. `XSetStandardProperties` is called to set the window title. XVT's other window manipulation functions (`move_window`, `set_title`, etc.) also map smoothly into X. XVT's double-line border style has no direct X equivalent, so an appropriate pixmap border is used instead.

To XVT the notions of frontmost window and holding the input focus are tied together, so its `set_front_window` call results in calls to both `XRaiseWindow` and `XSetInputFocus` on X.

Events—XVT has only 15 event types, most of which have direct equivalents in X, as shown in this table:

XVT Event	*Explanation*	*X Event*
E_MOUSE_DOWN	mouse button down	ButtonPress
E_MOUSE_UP	mouse button up	ButtonRelease
E_MOUSE_MOVE	mouse moved	MotionNotify
E_MOUSE_DBL	mouse button double-clicked	[see text]
E_CHAR	keyboard character typed	KeyPress
E_UPDATE	window needs updating	Expose, GraphicsExpose
E_ACTIVATE	window gets/loses focus	EnterNotify, LeaveNotify, FocusIn, FocusOut
E_KILL_WINDOW	last event for closed window	[internal]
E_VSCROLL	vertical scroll bar action	[internal]
E_HSCROLL	horizontal scroll bar action	[internal]
E_COMMAND	menu command	[internal]
E_CLOSE	request to close window	DestroyNotify
E_SIZE	window being resized	ConfigureNotify
E_FONT	selection from font/style menu	[internal]
E_QUIT	request to quit application	[not used]

Several XVT events (`E_KILL_WINDOW`, `E_HSCROLL`, `E_VSCROLL`, `E_COMMAND`, and `E_FONT`) are generated internally, and don't require direct support from the underlying window system. For example, an `E_COMMAND` event is generated when the user chooses a menu command. Since the menu manager is inside of XVT, it simply passes the event on to the application. Of course, the menu manager handles lots of events while the user is interacting with the menu, but those are hidden from the application.

The E_QUIT event only occurs on those systems that use a two-phase shut-down: Applications are first polled for approval, and then ordered to terminate if all are willing to do so. X doesn't use that protocol, so the applications's E_QUIT code, while portable, isn't actually executed under X.

The double-click event (E_MOUSE_DOUBLE) isn't supported directly by X (or by the Mac), but it's easily simulated because each button event includes a position and time. A double click is inferred when a button press follows a button release within a sufficiently small interval of time and area of the screen. The code to make this inference was taken from the Mac version of XVT. Note that if the time of the button event weren't passed in from the X server it would be infeasible to infer double clicks in the client, because time measurements there don't correspond accurately to the user's actions.

When XVT passes an E_UPDATE to an application it doesn't include information about what part of the window needs updating. Instead, the application calls the Boolean function needs_update to test each candidate rectangle (usually bounding boxes of graphical objects or blocks of text). XVT automatically sets the clipping region to the part that needs updating, so the application can just draw the entire window if it wants to, without calling needs_update at all. For the Mac and Windows, XVT's job is easy because those systems coalesce pending update events into a single event and arrange for the proper clipping. X's support for exposure events is at a lower level, so the X version of XVT has to take all of the exposure events from the queue (the last one is marked) and merge their rectangles (passed in the XExposeEvent structure) into a region. XVT then sets the clipping rectangle to the smallest box enclosing this region (with XClipBox and XSetClipRectangles) and calls XRectInRegion when needs_update is invoked.

Graphics—XVT uses a pen and brush model to draw shapes; the pen specifies the perimeter, and the brush describes the interior. These correspond to X's line and fill style in a graphics context. XVT doesn't maintain a graphics context as an object separate from a window, so to an XVT programmer it is the window's pen and brush that are set.

In XVT a single call (e.g., draw_rect) is used to draw both a filled and unfilled shape. If only the perimeter is wanted, a hollow brush is specified. For the X implementation, each shape routine tests the brush and calls the perimeter or filling variant of the appropriate X function. For example, if the brush is hollow, XDrawRectangle is called; if not, XFillRectangle is called. The Mac implementation uses the same method.

All but two of the XVT shapes map directly into X shapes. The exceptions are draw_roundrect and draw_pie (which draws a wedge). Both are built up in pieces by calling the X functions XDrawSegments and XDrawArcs (and their corresponding filling variants).

The XVT function draw_picture draws a picture that's normally stored as a collection of discrete objects, rather than a bitmap. This permits the picture to be rendered at the full resolution of the output device rather at the resolution of the device on which it was created. Both the Mac and Windows support such pictures (PICTs and Metafiles, respectively), but X does not. As a placeholder, we've used pixmaps to implement pictures—we'll implement our own object-oriented structure when time permits or, if an industry standard format emerges, we'll use that. Note that the obvious candidate here, PostScript, won't work because only those X implementations that support Display PostScript can draw such an object on the screen.

Fonts and Text—Selection of a font in which to output text is one of the most difficult XVT features to virtualize. The problem is that in most window systems the important attributes of a font are specified by a name (e.g., "fr3-25") which is meaningless in and of itself. Only MS-Windows has a font mapper that can select a font based on its characteristics, such as serif or sans-serif, weight, size, quality, and so on.

XVT sidesteps this problem by defining a custom font and style menu for each system on which it runs. The contents of this menu are hidden from the portable application; when the user makes a selection, an E_FONT event is passed to the application along with an abstract FONT object (whose internals are secret). To draw in that font, the FONT is passed as an argument to the function set_font. This scheme works fine for fonts that the user selects, but it doesn't work for fonts that the application wants to pick. That is, XVT provides no way to synthesize a locally meaningful FONT object from a list of portable attributes. Some of the most common needs are met by three predefined FONTs, big_font, normal_font, and small_font. We're working on a more general solution to the problem of synthesizing fonts that will probably employ a font mapper of our own design.

Meanwhile, the standard XVT font menu for X contains several of the most common font names. The user can change the list by editing the `.Xdefaults` file.

Cursors and Carets—XVT offers a choice of five predefined cursor shapes: arrow, I-beam, cross, plus (thicker cross), and waiting. All of them (and lots more!) exist in X's cursor font and can be set with a call to `XCreateFontCursor`.

In XVT terms a caret is a blinking line or solid block that indicates the insertion point for text. An application positions the caret with a call to `caret_on` and suppresses it with a call to `caret_off`. MS-Windows supports carets directly and handles the blinking automatically. The Mac doesn't, so XVT blinks it itself by XORing a small rectangle at fixed intervals of time. This works—in the grand tradition of PCs—because the Mac is a single-user machine with only rudimentary multitasking (the foreground application gets all the CPU cycles it wants). X lacks support for carets and one can't be blinked accurately or efficiently from a client process, so a non-blinking inverse rectangle is used instead. This works fine for character-oriented programs like Xterm, but WYSIWYG systems prefer to put the caret between the letters, not over a letter. A thin caret that doesn't blink is too hard for the eye to pick up rapidly. We're still pondering what to do about this.

Resources—Resources on the Mac and Windows are used more generally than those in X, primarily because memory is usually tight on those systems and loadable-on-demand resources provide an efficient place to store strings, menu definitions, dialog box layouts, and the like. Both systems come with elaborate dialog editors that allow an application designer (not necessarily a programmer) to create a dialog layout interactively. The design is then stored as a resource along with the executable image and is accessed at run time by a dialog manager. Aside from economizing on memory, resources also allow for late binding, which makes it easy to alter the design of a menu or dialog or to translate it into a foreign language.

We've designed a simple resource manager for X that supports just the XVT resource-related functions. These resources aren't kept in the `.Xdefaults` because that's under the user's control, and XVT's resources are part of the application. The same resource manager is used in the character-based version of XVT mentioned earlier.

User-Interface Toolkit Level

XVT's facilities in this layer support menus, dialog boxes, the clipboard, and printing. Only the clipboard is directly supported by X itself. Various toolkits have been developed to run on top of X to support menus and dialogs.

We had two constraints on our design of the X version of XVT that affected our choice of a toolkit:

- The toolkit must be universally available, wherever X is.
- It must be supported by the workstation vendor (certainly not by us!).

When we set down our X-11 plans in September, 1988, only the Xt toolkit met these criteria, but even it was much less robust than we wanted. Several promising toolkits were on the horizon, such as Open Look, DEC Windows, Sony Widgets, and Andrew, but none of them were available to us or to all but a few X users. We declined to postpone our port of XVT or to get involved with installing and debugging a public-domain toolkit ourselves.

We decided to pursue three lines of development. First, we would implement as much of XVT as Xt could handle, and make that available to our customers as a subset implementation. Second, we would design our own printer driver. Third, we would investigate implementing our own toolkit to support just the features that XVT needs. Actually, we had other reasons for doing this, because we also needed a character-based toolkit.

We consider these initial toolkit projects to be a stopgap only, so we didn't want to put too much effort into them, and we didn't worry about the esthetics of the user interface. We plan to produce a final version of XVT for X as soon as the chaotic toolkit area settles down. That version will most likely use AT&T's Open Look X-based toolkit. There will be another port for the toolkit that the Open Software Foundation chooses if that turns out not to be Open Look. We'll begin those ports when these toolkits are up and running in our shop—it's too hard for us to start earlier (we already tried using the viewgraphs in our code, but the linker complained).

The next few paragraphs discuss some specifics in our design of the XVT toolkit level for X.

Menus—XVT needs very little in the way of menus. It uses a two-level menu hierarchy inspired by the Mac menu bar, although the top level need not be a bar. We're implementing a simple menu manager patterned after the deck-of-cards menu package (Xmenu) that was shipped with X-10.

The Mac and Windows menu managers also handle keyboard shortcuts that allow a function key or control-key combination to be used as a synonym for a menu item. However, how the command is invoked is hidden from an XVT application, so there was no requirement to implement that feature for X right away, and we put it off.

Dialogs—XVT expects the system on which it runs to support a sophisticated dialog manager that takes over user interaction when a dialog is entered. It calls an application-designated callback function whenever the user does anything notable (such as pressing a button). Dialogs themselves are described by *control lists* of *control items*. Each item consists of a type code (button, check box, edit field, etc.), bounding rectangle, text string, activation flag, and a few other attributes. When a dialog is brought up the dialog manager reads the control list from the resource file and builds a displayable box containing the specified controls. It then supervises the user's interaction while the focus is in the box.

As of December, 1988, we were leaning against using Xt widgets to handle dialogs. We had already begun implementing a character-based dialog manager that had all the right semantics; adapting it for a graphical world would involve only replacing the rendering and action functions for each control type (button, radio button, check box, static text, editable text, scroll bar, and list box). We knew that our own design would work, but we also wanted to use the Xt system because it would give us a leg up on using the commercial toolkits later. The chief problem with Xt is that the predefined widgets (from MIT) handle only a few of the controls that XVT needs. With our limited resources we were reluctant to get involved in designing our own widgets.

We may end up initially with only a partial implementation of dialogs for X, with the intention of filling in the gaps later. Indeed, even the original XVT implementation for the Mac and Windows lacked scroll bar and list box control types.

Clipboard—XVT supports the placing of one object on the clipboard, in as many formats as the application desires. Ideally, an application should be willing to put both text and picture formats onto the clipboard, and to take either off. An application can also use one or more private formats, available only to itself or to other instances of itself. It's OK for an application to use only text or only pictures if one of them isn't meaningful (e.g., an electronic mail program might not handle pictures).

These facilities are supported by X itself, so no toolkit support is needed. X's selection mechanism works well, except for the lack of a standard object-oriented picture format. As mentioned earlier, we decided to use pixmaps temporarily.

Printing—Printing was our single biggest challenge in moving XVT to X. On the Mac, in Windows, and in XVT one prints by opening a special, non-displayable print window and then drawing on it with the identical drawing functions used for normal windows. It's extremely easy to print a document (containing text and/or graphics) that the user has composed in a screen window: You just open a print window and make like you got an update event for it.

Unfortunately, X doesn't work that way. It doesn't support printing at all, although most X implementations allow you to print the screen or the contents of a window. But you want 1200 dots-per-inch on your typesetter, not seventy-five!

We had no choice but to implement our own printer driver. To make the job tractable, we did limit it to PostScript printers only. The interface is documented so that others can implement additional drivers. If a proper printing facility is ever added to X (perhaps part of one of the commercial toolkits), we'll just implement a "driver" that uses the standard printer interface and throw away our real driver.

Here's how our simple design works: In each XVT drawing call, such as `set_pen` or `draw_oval`, we branch according to whether the current window is a screen window or a print window. In the former case, we call the regular X functions. In the latter case we call a driver entry point. These are named after the corresponding XVT calls: `pr_set_pen`, `pr_draw_oval`, etc. Our PostScript driver simply issues appropriate PostScript code to perform each action. Debugging this driver is extremely easy with a utility called Lasertalk running on a Mac connected to a LaserWriter. Lasertalk allows you to view printer output in a window

on the screen, without printing anything. The output is guaranteed to be authentic because the PostScript is executed in the printer itself. Debugging would have been even easier with Display PostScript, but, alas, we didn't have access to it.

Not Yet the Conclusion—The XVT implementation on X will work well enough to demonstrate the viability of our approach, but its user interface isn't quite ready for prime time.

Support in X and UNIX at the operating system and window levels is good—no major problems there. The user-interface toolkit level is still confused. We've taken some temporary steps to get XVT running, but we knew from the start that we would recode most of it later in 1989 or 1990 once the toolkits stabilize. Meanwhile, XVT will indeed run on X, and the application interface is nearly 100% supported. So our immediate goal has been realized: It is now possible to write a portable program just once and run it on a Macintosh, an IBM-compatible running Windows or Presentation Manager, or a UNIX-based workstation running X.

REFERENCES

[Nye88] Nye, Adrian (ed.). *Xlib Programming Manual.* Newton, MA: O'Reilly & Associates Inc., 1988. ISBN 0-937175-27-7

[Roc87] Rochkind, Marc J. *Technical Overview of the Extensible Virtual Toolkit (XVT).* Boulder, CO: Advanced Programming Institute Ltd.

[Roc88] Rochkind, Marc J. *Advanced C Programming for Displays.* Englewood Cliffs, NJ: Prentice-Hall, 1988. ISBN 0-13-010240-7

APPENDIX: EXAMPLE XVT PROGRAM

The following is an example XVT program that can display either "Hello World!" or "Goodbye World" in one of several windows opened by the user. The message can be chosen via the Choice menu, by double-clicking the mouse, or by typing a "g" or an "h." The font and style for each window can be independently chosen. Figure 2 (appearing in the body of this paper) shows a sample session.

As discussed in the paper, this program is portable without change to each environment on which an XVT library is implemented.

```
#include "xvt.h"                                /* standard XVT header */
#include "xvtmenu.h"                            /* standard XVT menu tags */

#define M_CHOICE_HELLO    MAKE_MENU_TAG(MENU3, 1) /* tags for our Choice menu */
#define M_CHOICE_GDBYE    MAKE_MENU_TAG(MENU3, 2)

#define STARTX            30                    /* x coordinate for message */
#define STARTY            40                    /* y coordinate for message */

typedef struct {
     enum {HELLO, GOODBYE} choice;              /* message to display */
     FONT font;                                 /* font */
} DOCUMENT;                                     /* contents of one document */

APPL_SETUP appl_setup;                          /* setup for initial window */

/*
     Function to associate a DOCUMENT structure with a new window. Each window
     carries a long word of data that can be set and accessed by the application.
     We use it to hold a pointer to a "document," which for this simple example
     consists of only the message choice and its font and style. More generally,
     a document structure would hold text, a picture, etc.
*/
static void new_doc(win)
WINDOW win;                                     /* window to be initialized */
{
     DOCUMENT *d;
```

```
     if ((d = (DOCUMENT *)malloc(sizeof(DOCUMENT))) == NULL)
          fatal("Out of memory.");
     d->choice = HELLO;
     d->font = normal_font;
     set_app_data(win, PTR_LONG(d));
}

/*
     Function to destroy a document structure, to be called when a window is
     closed (assuming that a document appears in only one window).
*/
static void dispose_doc(d)
DOCUMENT *d;                                        /* document to be destroyed */
{
     free((char *)d);
}

/*
     Function called by XVT to initialize application. It selects the font for the
     initial window (std_win), makes sure that it's marked appropriately in the
     Font/Style menu, initializes the window's document, sets a check mark in the
     Choice menu, and enables the New item on the File menu.
*/
BOOLEAN appl_init()
{
     set_cur_window(std_win);
     set_font(&big_font, FALSE);
     set_font_menu(&big_font);
     new_doc(std_win);
     menu_check(M_CHOICE_HELLO, TRUE);
     menu_enable(M_FILE_NEW, TRUE);
     return(TRUE);
}

/*
     Function to handle update events. The screen is painted white and then the
     chosen message is drawn inside a rounded rectangle. To fit the shape
     properly, the message is first measured in width and height, and a rectangle
     is initialized that fits the text with a margin all around that's equal to
     the descent. (Nothing magical about the descent - it's just a convenient
     amount that's proportional to the size of the font.)
*/
static void do_update(win)
WINDOW win;                                         /* window to be updated */
{
     RCT rct;
     char *msg;
     int ascent, descent, width, corner;
     DOCUMENT *d = (DOCUMENT *)get_app_data(win);

     get_client_rect(win, &rct);
     set_pen(&white_pen);
     set_brush(B_WHITE);
     draw_rect(&rct);
     msg = d->choice == HELLO ? "Hello World!" : "Goodbye World!";
     set_font(&d->font, FALSE);
     get_font_metrics(NULL, &ascent, &descent);
     width = get_text_width(msg, -1);
     set_rect(&rct, STARTX - descent, STARTY - descent, STARTX + width + descent,
       STARTY + ascent + 2 * descent);
     corner = (rct.bottom - rct.top) / 3;
     set_pen(&black_pen);
     draw_roundrect(&rct, corner, corner);
     draw_text(STARTX, STARTY + ascent, msg, -1);
}
```

```
/*
     Function to set check marks on the Choice menu.
*/
static void fix_choice_menu(d)
DOCUMENT *d;                                    /* active document */
{
     menu_check(M_CHOICE_HELLO, d->choice == HELLO);
     menu_check(M_CHOICE_GDBYE, d->choice == GOODBYE);
}


/*
     Function to handle menu commands.
*/
static void do_menu(cmd)
MENU_TAG cmd;                                   /* tag identifying menu item */
{
     char title[50];
     RCT rct;
     static int count = 0;
     WINDOW win;
     DOCUMENT *d;

     switch (cmd) {
     case M_FILE_NEW:
          set_rect(&rct, 100, 100, 300, 200);
          sprintf(title, "Window %d", ++count);
          if ((win = new_window(&rct, title, W_DOC, TRUE, FALSE, FALSE, TRUE)) ==
            NULL)
               error("Can't create window.");
          new_doc(win);
          break;
     case M_FILE_QUIT:
          terminate();
          break;
     case M_CHOICE_HELLO:
     case M_CHOICE_GDBYE:
          if ((win = get_front_window()) != NULL) {
               d = (DOCUMENT *)get_app_data(win);
               d->choice = cmd == M_CHOICE_HELLO ? HELLO : GOODBYE;
               fix_choice_menu(d);
               invalidate_rect(win, NULL);
          }
     }
}

/*
     Main application entry point. After changing the message choice or the font,
     the entire client area of the window is invalidated so as to force an update.
     On a font event (user selected from the Font/Style menu) the font associated
     with the document is changed and the menus themselves are modified to show
     the current font.

     An E_QUIT event is generated by the system when it wants to shut down. In
     phase one, each running application is polled as to whether it is willing
     to quit (our little application is, so it calls quit_OK). If the vote was
     unanimously positive, in phase two each application is ordered to quit. If
     any application declined in phase one, phase two is skipped and the system
     stays up. (Not all window systems use this protocol.)
*/
void main_event(win, ep)
WINDOW win;                                     /* window receiving event, if any */
EVENT *ep;                                      /* pointer to event structure */
{
     DOCUMENT *d = NULL;
```

```
        if (win == NULL)
             win = get_front_window();
        if (win != NULL)
             d = (DOCUMENT *)get_app_data(win);
        switch (ep->type) {
        case E_MOUSE_DBL:
             if (d->choice == HELLO)
                  d->choice = GOODBYE;
             else
                  d->choice = HELLO;
             invalidate_rect(win, NULL);
             break;
        case E_CHAR:
             if (d != NULL) {
                  switch (toupper(ep->v.chr.ch)) {
                  case 'G':
                       d->choice = GOODBYE;
                       break;
                  case 'H':
                       d->choice = HELLO;
                  }
                  invalidate_rect(win, NULL);
             }
             break;
        case E_UPDATE:
             do_update(win);
             break;
        case E_ACTIVATE:
             if (ep->v.active && d != NULL) { /* make menu state reflect document */
                  set_font_menu(&d->font);
                  fix_choice_menu(d);
             }
             break;
        case E_COMMAND:
             do_menu(ep->v.cmd.tag);
             break;
        case E_FONT:
             if (d != NULL) {
                  d->font = ep->v.font.font;
                  set_font_menu(&ep->v.font.font);
                  invalidate_rect(win, NULL);
             }
             break;
        case E_CLOSE:
             if (d != NULL)
                  dispose_doc(d);
             close_window(win);
             break;
        case E_QUIT:
             if (ep->v.query)
                  quit_OK();
             else
                  terminate();
        }
   }

   /*
        Function called by XVT just before terminating application.
   */
   void appl_cleanup()
   {
        /* no cleanup (e.g., files to be removed) needed */
   }
```

Viral Attacks On UNIX® System Security

Tom Duff

AT&T Bell Laboratories
Murray Hill, NJ
research!td
td@research.att.com

ABSTRACT

Executable files in the Ninth Edition of the UNIX system contain small amounts of unused space, allowing small code sequences to be added to them without noticeably affecting their functionality. A program fragment that looks for binaries and introduces copies of itself into their slack space will transitively spread like a virus. Such a virus program could, like the Trojan Horse, harbor Greeks set to attack the system when run by sufficiently privileged users or from infected set-userid programs.

The author wrote such a program (without the Greeks) and ran several informal experiments to test its characteristics. In one experiment, the code was planted on one of Center 1127's computers and spread in a few days through the Datakit® network to about forty machines. The virus escaped during this test onto a machine running an experimental secure UNIX system, with interesting (and frustrating for the system's developers) consequences.

Viruses of this sort must be tiny to fit in the small amount of space available, and consequently are very timid. There are ways to construct similar viruses that are not space-constrained and can therefore spread more aggressively and harbor better-armed Greeks. As an example, we exhibit a frighteningly virulent portable virus that inhabits shell scripts.

Viruses rely on users and system administrators being insufficiently vigilant to prevent them from infiltrating systems. I outline a number of steps that people ought to take to make infiltration less likely.

Numerous recent papers have suggested modifications to the UNIX system kernel to interdict viral attacks. Most of these are based on the notion of 'discretionary access controls.' These proposals cannot usually be made to work, either because they make unacceptable changes in the 'look and feel' of the UNIX system's environment or they entail placing trust in code that is inherently untrustworthy. In reply to these proposals, I suggest a small change to the UNIX system permission scheme that may be able to effectively interdict viral attacks without serious effect on the UNIX system's functioning and habitability.

Introduction

UNIX system security has been a subject of intense interest for many years, in large part because the UNIX system is more secure than any other operating system offering comparable facilities and as such offers a more challenging target for recreational breaking-and-entering. The *ne plus ultra* of UNIX system security breaking is to have the super-user execute arbitrary code on behalf of the miscreant. The most common way to do this is to find a root-owned set-userid program that calls the shell and exploit any of a number of well-known loopholes to get

it to execute a chosen command file. [Reeds] describes a number of variations on this theme.

Other interesting possibilities are to convince someone who has write permission on a root-owned set-userid program to modify it to execute chosen code, or to get someone running as super-user to run code provided by the miscreant. No responsible individual would do such a thing deliberately. [Thompson] describes an extremely clever surreptitious way of doing the former; [Grampp & Morris] discuss ways of getting the unwary super-user to do the latter.

The likelihood of the super-user inadvertently executing miscreant-supplied code is a function of the number of files that contain copies of the code. A program could be written that would try to spread itself throughout the file system by searching for writable binary files and patching copies of itself into them. It would have to be careful to preserve the functionality of the modified programs, in order to avoid detection. Eventually it might so thoroughly infect executable files that it would be very unlikely for the super-user never to execute it.

A Virus For UNIX System Binaries

Ninth Edition VAX UNIX system files containing executable programs start with a header of the following form, inherited from 4.1 BSD, of which the Ninth Edition is a descendent:

```
struct {
        int magic;              /* magic number */
        unsigned tsize;         /* size of text segment */
        unsigned dsize;         /* size of data segment */
        unsigned bsize;         /* size of bss segment */
        unsigned ssize;         /* size of symbol table */
        unsigned entry;         /* entry point address */
        unsigned trsize;        /* size of text relocation */
        unsigned drsize;        /* size of data relocation */
};
```

If the magic number is 413 in octal, the file is organized to make it possible to page the text and data segments out of the executable file. Thus the first byte of the text segment is stored in the file at a page boundary, and the length of the text segment is a multiple of the page size, which on our system is 1024 bytes. Since a program's text will only rarely be a multiple of 1024 bytes long, the text segment is padded with zeros to fill its last page.

With this in mind, the author wrote a program called `inf` (for *infect*) that examines each file in the current directory. Whenever `inf` finds a writable 413 binary with enough zeros at the end of its text segment, it copies itself there, patches the copy's last instruction to jump to the binary's first instruction, and patches the binary's entry point address to point at the inserted code. `Inf` is only 331 bytes long. All other things being equal, it has about two chances in three of finding enough space to copy itself into a given binary.

Once a system is seeded with a few copies of the virus, and with a little luck, someone will sooner or later execute one of the modified binaries either from a different directory or from a userid with different permissions, spreading the infection even farther. Our UNIX systems are connected by a network file system [Weinberger], so there is a good chance of the infection spreading to files on other machines. We also have an automatic software distribution system [Koenig], intended to keep system software up-to-date on all our UNIX systems. Even wider distribution is possible with its aid.

Spreading The Virus

I tried a sequence of increasingly aggressive experiments to try to gauge the virus's virulence. Many users leave general write permission on their private `bin` directories. So, on May 22, 1987 I copied `inf` into /usr/*/bin/a.out on Arend, one of Center 1127's VAX 11/750s. My hope was that eventually someone would type a.out when no such file existed in their working directory, and my program would quietly run instead.

Unsurprisingly, this hope proved fruitless. By July 11 `inf` had spread not at all, except

amongst my own files, where it had gotten loose accidentally during testing. Only one of Arend's regular users other than myself got a copy of the program, and that was never executed. It should be noted that while nobody got caught, neither did any of the 14 people whose directories were seeded notice that anything was awry.

With the failure of this extremely timid approach, on July 11 I infected a copy of /bin/echo and left the result on Arend in /usr/games/echo and /usr/jerq/bin/echo – two directories on which I had write permission, and which I had observed several users to search before /bin. I supposed that one of these users would eventually run echo, infect a few files and we'd be off to the races. This in fact happened three times (on July 21, July 30 and August 7), infecting four more files. By September 10, the infection had spread no farther.

On September 10, I attacked Coma, a VAX 8550, far and away the most-used machine in our center. I looked in /usr/*/.profile to see what directories someone searched before /bin, and placed infected copies of echo in the 48 such directories that I could write. The infection spread that day to 11 more files on Coma, and a further 25 files on the following day, including a newly compiled version of the wc(1) command. The infected /bin/wc was distributed to 45 systems by the automatic software distribution system [Koenig]. The experiment was stopped on September 18, at which time there were 466 infected files on the 46 systems.

Only four of the 48 users who were seeded noticed that their directories had been tampered with and asked what was going on. All seemed satisfied with explanations of the form "yes, I put it there" or "I'll tell you later." In any case, none of them felt a need to remove the file.

One of the machines infected by the virus was Giacobini, a machine being used by Doug McIlroy and Jim Reeds to develop a multi-level secure version of the Ninth Edition UNIX system that retains as much of the flavor of standard insecure UNIX systems as possible. Probably they accepted the automatic distribution of the infected wc command. They did not, however, accept shipment of the 'disinfect' program that put an end to the experiment, so `inf` lived on and continued to spread on their machine. On October 14 they turned on their security features for the first time and soon thereafter discovered programs dumping core because of security violations that should not have occurred. Here is Jim Reeds' account of the virus's effect on their system and how they eventually excised it:

> From reeds Fri Oct 16 11:20 EDT 1987
>
> Not sure how the virus got on giaco. Maybe via asd, maybe placed as a gentle prank, possibly a long dormant spore. Maybe even it was there all along, infesting up everything, and the new security stuff made it visible. Dozens of files were infected: ar, as, bc, ... most of the files in the public bins, my private bin directory, and a couple in /lib. When I cottoned on to what was happening I went on a disinfect frenzy, muddying up modification dates that would have helped in figuring out where it came from. It got a private su command of mine, so it started spreading with root privs in /etc. After a while every command I typed took a couple of seconds longer than it should have. Df, for instance, takes a fraction of a second per line, now seemed to take several seconds per line. I thought it was the security stuff bogging the system down. But what really vexed me was this: whenever I tried to run my su command when I was in /etc the command died after a pause. Hours later, & kernel printfs galore, it transpired that it always died because it tried to write on file descriptor 5 which was attached to /etc/login, which earlier in the day I had marked as 'trusted', which means absolutely nobody may write on it. I proceded on the theory that I had a kernel bug (not new to me these last weeks, mind you) that gave such a wrong file descriptor. Finally I had narrowed the 'bug' down to happening when this program
>
> ```
> .word  0
> chmk   $1
> ```
>
> was assembled and linked 413 and executed out of /etc. Then I began to smell

a rat. Comparison with binaries on other machines, discovery of ´disinfect', disassembly, blah blah blah. Because I was doing heavy (for me) kernel hacking I was sure kernel bugs explained all anomalous behavior.

In all it took 1.5 working nights to figure it out. During the last 1/2 day or so performance took a nose dive: a make in the background and giaco was like alice on a busy day. I guess this recent performance hit argues against the virus having been active for a long time.

More Vigorous Viruses

`Inf` is not very virulent, and its only insalubrious effect is the mild system degradation that its execution causes. This is a consequence of a desire to keep the size of the program down to maximize the number of binaries it would fit in. Placing a Greek in this Trojan Horse would be easy enough. For example, in a few instructions we could look to see if the program's argument count is zero, and if so execute /bin/sh. This test is unlikely to succeed by accident. It's impossible for the shell to execute a command with a zero argument count since, by convention, the first argument of any command is the command name. But the following simple program has the desired effect:

```
main(){
        execl("infected_a.out", (char *)0);
}
```

If the infected program is set-userid and owned by root, this will give the miscreant a superuser shell.

`Inf` can add noticeably to the execution time of infected programs, especially in large directories. This could be fixed by having the virus fork first, with one half propagating itself and the other half executing the code of the virus's host.

The virus's small size seriously restricts its actions. A virus that looked at more of the file system could certainly spread itself faster, but it's hard to imagine fitting such a program into little enough space that it would find places to fit itself. The size limitation can be overcome by expanding the victim's data segment to hold the virus. After executing, the virus would have to clean up after itself, setting the program break to the value expected by the victim, and clearing out the section of the expanded data segment that the host was expecting to be part of the all-zero bss segment. After it's zeroed itself, the virus must jump to the first instruction of its host. This seems tricky, but it should be possible to do it by copying the cleanup code into the stack.

The virus is further restricted by being written in VAX machine language. It therefore cannot spread to machines with non-VAX CPUs or even to machines that run incompatible variants of the UNIX system. A virus to infect Bourne shell scripts would be insensitive to the cpu it ran on, and could be made fairly insensitive to different versions of the UNIX system with a little care. Here is the text of a virus called `inf.sh` that should be portable to most contemporary versions of the UNIX system:

```
#!/bin/sh
(     for i in * /bin/* /usr/bin/* /u*/*/bin/*
      do    if    sed 1q $i | grep '^#![      ]*/bin/sh'
            then  if    grep '^# mark$' $i
                  then  :
                  else  trap "rm -f /tmp/x$$" 0 1 2 13 15
                        sed 1q $i >/tmp/x$$
                        sed '1d
                              /^# mark$/q' $0 >>/tmp/x$$
                        sed 1d $i >>/tmp/x$$
                        cp /tmp/x$$ $i
                  fi
```

```
        fi
    done
    if  ls -l /tmp/x$$ | grep root
    then rm /tmp/gift
         cp /bin/sh /tmp/gift
         chmod 4777 /tmp/gift
         echo gift | mail td@research.att.com
    fi
    rm /tmp/x$$
) >/dev/null 2>/dev/null &
# mark
```

`Inf.sh` examines files that start with `#!/bin/sh` in a number of likely directories and copies itself into each one that doesn't appear already to be infected. `Inf.sh` contains a Greek that places a set user-id shell in `/tmp/gift` and mails me notification whenever the virus appears to be running as super-user.

However sorely you are tempted, *do not* run this code. It got loose on my machine while being debugged for inclusion in this paper, and within an hour had infected about 140 files, at which time several copies were energetically seeking other files to infect, running the machine's load average, normally between .05 and 1.25, up to about 17. I had to stop the machine in the middle of a work day and spend three hours scouring the disks, earning the ire of ten or so co-workers. I feel extremely fortunate that it did not escape onto the Datakit network.

Countermeasures

Spreading a virus has several requirements. First, the virus must have a way of making viable copies of itself. Second, the miscreant must have a way to place seed copies of the virus where they will be executed. Third, the infection must be hard for system administrators to spot. All of these requirements are relative. A particularly virulent virus might be easy to spot and yet be successful because it can spread faster than anyone might notice.

UNIX system administrators and users can take many measures to limit the danger of viral attack.

- Do not put generally writable directories in your shell search path. These are prime places for a miscreant to seed.
- Beware of Greeks bearing gifts. Imported software should carefully be examined before being loaded onto a sensitive machine. In the best situation you will have all source code available to read and understand before compiling it with a trusted compiler. In the absence of source code it is also helpful to have a controlled environment in which to exercise the code before letting it loose on trusted machines. The ideal test environment would be a machine that can be disconnected from all communications equipment and whose storage media (disks, tapes, Williams tubes, etc.) can be reformatted and reloaded with old data if any infection appears. Ideal conditions often do not obtain. You should try your best to approximate them as closely as possible with the resources available to you.
- Watch for changing binaries. System administrators should regularly check that all files critical to the daily operation of the system do not change unexpectedly. The most complete way to do this would be to maintain copies of all critical files on read-only media and periodically compare them with the active copies. Most systems will not have such media available. An adequate compromise is to maintain a list of checksums and inode change dates (printed by `ls -lc`) of the critical files. The inode change date is updated whenever the file is written and is difficult to set back without either patching the disk or resetting the system clock. The checksum function should be hard to invert, to thwart viruses that try to modify themselves in a way that preserves the checksum. Hard-to-invert functions are called one-way functions in the cryptographic literature. Encrypting the file using DES in

cipher-block chaining mode and using the last block of ciphertext as the checksum is probably a good one-way checksum.

- Our automatic software distribution system [Koenig] is a wonderful tool for keeping software up-to-date amongst a collection of machines. It is also a powerful vector for transmitting viruses. The wide and rapid spread of `inf` can largely be attributed to its inadvertently having been distributed to all of our machines hidden in a copy of the `wc` command. People who distribute software should be careful that they only ship newly compiled, clean copies of their code. Versions that have been used for testing may very well have been infected.

System Enhancements to Interdict Viruses

There are several proposals in the literature to stop the spread of viruses by what are called 'discretionary access controls.' This buzzword indicates a system organization in which all of a program's accesses to files are authorized by the user running the program. [Lai & Gray] point out that users cannot reasonably be expected to explicitly authorize all file accesses, or they would continually be interrupted by innumerable queries from the kernel. They suggest dividing binaries into two camps, trusted and untrusted. The word 'trust' here has a different meaning than in McIlroy and Reeds's secure UNIX system, discussed above. Trusted binaries, like the shell and the text editor, are allowed access to any file, subject to the normal UNIX system permission scheme. When an untrusted binary is executed by a trusted one, it may access only files mentioned on its command line. If the untrusted binary executes any binary, the new program is invariably treated as untrusted (even if it has its trusted bit set) and inherits the set of accessible files from its parent. (Lai and Gray make other provisions to allow suites of untrusted programs to create temporary files and use them for mutual communication, but those provisions are irrelevant to our discussion.)

Among the underlying assumptions of Lai and Gray's scheme are that users do not ordinarily write programs that would require trusted status, and that the system programs that require trusted status (they name 32 binaries in 4.3BSD that require trust) are actually incorruptible. Neither assumption is justifiable. Perhaps there is a class of casual programmers that will be satisfied writing programs that can only access files named on the command line, but it is hard to imagine software of any complexity that does not include editing or data management facilities that are ruled out by this scheme. A user cannot even, as is common, write a long-running program that sends mail to notify the user when it finishes, because /bin/mail is one of the system programs that requires trust, and when executed from an untrusted program it will not have it.

The assumption of incorruptibility of trusted programs is equally unjustified. The `inf.sh` virus or a slight variant of it would spread uncontrolled under Lai and Gray's scheme, because it will be executed by a shell running in trusted mode.

Lai and Gray's scheme does not go far enough, as it does not effectively interdict the behavior that it attacks. Simultaneously it goes too far, altering the UNIX environment beyond recognition and rendering it unusably clumsy. The only possible conclusion is that they are going in the wrong direction.

I see no way of throwing out Lai and Gray's bathwater and keeping the baby. Any scheme that requires that the shell be trusted entails crippling the shell. Users that are unsatisfied with the crippled shell are prevented from replacing it, since the replacement cannot have the required trust. This is an unacceptable violation of the precept that the entire user-level environment be replaceable on a per-user basis. [Ritchie & Thompson]

Modifying UNIX system file protection to interdict viruses

Having attacked one suggested virus defense, it is with some trepidation that I suggest another. The UNIX system uses a file's execute permission bits to decide whether the `exec` system call ought to succeed when presented with a file of the correct format. The execute bits are nor-

mally set by the linkage editor if its output has no unresolved external references. This amounts to certification by the linkage editor that, as far as it is concerned, the binary is safe to execute. The rest of the system treats the execute bits as specifying permission rather than certification. The bits are settable at will by the file's owner, and are not updated when the file's content changes. As permission bits they are nearly useless; almost always executable files are also readable (in my search path there are 602 executable files, only one of which (/usr/bin/spitbol) is not also readable) and so can be run by setting the execute bits of a copy.

I propose changing the meaning of the execute permission bits so that they act as a certificate of executability, rather than permission. Under this scheme, when you see a file with its execute bits set, you should think "some authority has carefully examined this file and has certified that it's ok for me to execute it." The implementation will involve a few small changes to the kernel. First, changing a file will cause its execute bits to be turned off, as any previous certification is now invalid. The effect of this will be to stop a virus from its transitive self-propagation. In addition, users and system administrators will be alerted that something is awry when they notice that formerly-executable commands no longer are. Second, the group and others execute bits may only be set by the super-user, who is presumably an appropriate certifying authority, and in any case has more expedient means of causing mischief than malicious execute-bit setting. Logging any changes to executable files would aid in tracking down any viruses that try to attack the system.

In many open environments, the requirement that setting the group and other execute bits be restricted to the super-user will be regarded as too oppressive for the increment of security that it provides. In such cases, the chown command can easily be made root-owned and set-userid and modified to implement any appropriate policy.

Acknowledgements

Some of the ideas described here arose in conversations with Norman Wilson and Fred Grampp. Ron Gomes helped make the Bourne shell virus more portable.

References

[Grampp & Morris] F. T. Grampp and R. H. Morris, *UNIX Operating System Security*, AT&T Bell Laboratories Technical Journal, Vol. 63 No. 8 Part 2, October 1984, pp. 1649-1672

[Koenig] Andrew R. Koenig, *Automatic Software Distribution*, USENIX 1984 Summer Conference Proceedings, pp. 312-322

[Lai & Gray] Nick Lai and Terence E. Gray, *Strengthening Discretionary Access Controls to Inhibit Trojan Horses and Computer Viruses*, Proceedings of the Summer 1988 USENIX Conference, pp. 275-286

[Reeds] Jim Reeds, */bin/sh: the biggest UNIX security loophole*, AT&T Bell Laboratories Technical Memo, 1988

[Thompson] Ken Thompson, *Reflections on Trusting Trust*, Comm. ACM Vol 27 Number 8 (August 1984) pp. 761-763 (1983 Turing Award lecture)

[Ritchie & Thompson] D. M. Ritchie and K. Thompson, *The UNIX Time-Sharing System*, Comm. ACM, 17, No. 7 (July 1974), pp. 365-375.

[Weinberger] Peter J. Weinberger, *The Version 8 Network File System (abstract)*, USENIX 1984 Summer Conference Proceedings, pg. 86

A Faster *fsck* for BSD UNIX™

Eric J. Bina

Motorola Micro-Computer Division
Urbana Design Center

Perry A. Emrath

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign

Abstract

In the UNIX™ operating system, the kernel attempts to maintain the file system in a known correct state. Should the kernel detect variation from that state, a system panic occurs, and the system "crashes". For this reason the integrity of the file system must be checked before bringing any UNIX system up in multi-user mode. A file system check program called *fsck* is supplied with UNIX (unless otherwise specified UNIX refers to the BSD version family), and it finds errors in the state of the file system and fixes them, if possible, to prevent system panics.

Unfortunately, on systems with a large amount of mounted disk storage, the time it takes to run *fsck* can become a significant percentage of system boot time. When working in a system development environment where the system will probably be booted frequently, the (highly desirable) use of *fsck* can begin to waste valuable work time. To solve this problem, *fsck* was studied in detail, and modified to perform its tasks more efficiently.

This paper first describes the original *fsck* program to convey the goals it sets out to accomplish. The next section points out inefficiencies in the original algorithms, and describes how these inefficiencies were alleviated. Timing measurements were taken to locate the original problems, and to show that the modifications provided significant improvement. Our new *fsck* was tested to ensure it retained all the diagnostic and corrective capabilities of the original. It was also ported to a different machine to demonstrate its general usefulness in any BSD based UNIX environment.

Compared to the version of *fsck* distributed with BSD 4.2 or 4.3, our *fsck* consistently runs 2 to 3 times faster, and reboot times (with file system checks) have been cut almost in half. Real time measurements for our *fsck* on three machines where it has been installed are given in the concluding section.

This work was supported in part by the National Science Foundation under Grant No. NSF-MIP-8410110 and the U. S. Department of Energy under Grant No. DOE-DE-FG02-85ER25001.

1. Overview of the *fsck* Program

The *fsck* program examines all critical file system data and detects any inconsistencies or corruptions of that data. As a very important secondary goal, *fsck* attempts to correct any problems it finds. To find problems, *fsck* looks at the file systems organizing data structures. This means *fsck* must examine the super block, all inodes, the bit maps within the cylinder groups, and all data blocks belonging to directory files. It is assumed that the reader has some idea of the contents and purposes of these file system data structures. For descriptions of these entities, see [Bach 1986] and [McKusick 1984]. With information gained from these varied sources, as well as the knowledge of what a correct file system should look like, *fsck* determines the current state of the file system and what action if any should be taken to correct it.

The file system is checked by five passes through file system data. Pass 1 reads and checks all inodes and constructs a bit map of used data blocks. Pass 2 traverses and checks the structure of the directory tree. Pass 3 finds all directories that for some reason are disconnected from the directory tree, and reconnects them. Pass 4 checks the link counts for all inodes to make sure they are correct. Pass 5 checks the bitmaps and summary counts for each cylinder group, and corrects them if necessary. In addition, there is a pass that is only executed if pass 1 finds duplicate references to any data blocks. This pass, called pass 1b, attempts to locate all owners of duplicate blocks, so the user can decide on the proper owner. Also, there are some checks which are done in the preliminary set up of the *fsck* program.

Fsck begins by reading and checking the super block for the file system it is examining. Because the super block is vital, there exists a copy of the super block within each cylinder group; *fsck* checks that the real super block is identical to the copy stored in the last cylinder group. Only if the super block is correct will *fsck* go on to execute the passes. If the super block was incorrect *fsck* can be invoked with an option (*fsck -b*[block #]) that specifies an alternate block for the super block data.

In pass 1, *fsck* performs a number of checks, and collects data to be used in later passes. Critical among the data it collects are three maps. The first is a map of all data blocks that are in use in the file system. The second is an inode "state" map that marks whether each inode in the file system is unused, a file, or a directory. Third is a map that stores the link count field from each inode. The checks in pass 1 are done sequentially, starting with the first cylinder group in the file system. Within each cylinder group, pass 1 reads and checks all the inodes belonging to it. *Fsck* checks for partially allocated or corrupted inodes by ensuring that all fields in the inode contain reasonable values. Then *fsck* checks that the number of block pointers in the inode do not exceed its block count field. For any inode that fails these checks the user is prompted to see if the inode should be deleted. Pass 1 then stores the state and link count of the inode in the appropriate maps.

Checking is then done on each block number listed within the current inode to see if it is outside the allowed legal values for the file system. Inodes which point to illegal blocks are marked as bad in the inode state map, and in pass 4 the user can choose whether to keep or delete them. All claimed data block numbers are marked as used in the data block map. If any blocks are claimed by more than one inode they are noted, and the extra pass 1b will eventually be executed. Finally, all of the used blocks in the inode are counted, and this count is compared to the block count field. If the two block counts disagree the user can decide to replace the old value with the new value.

If there were any multiply claimed data blocks, pass 1b will now begin execution. Pass 1b will repeat the reading of all inodes that was done in pass 1. However, in pass 1b all references to duplicate blocks will be caught, instead of just the second one. Each such inode will be marked bad as if it referred to an illegal block number. Pass 4 will later allow the user to decide which if any of these inodes should be deleted. Usually, pass 1b is not executed since duplicate references to data blocks are rare.

In pass 2, *fsck* no longer looks at the file system as a flat set of inodes and data blocks, but instead treats it as a tree structure. Thus, pass 2 starts with the inode for the root directory and begins a depth first traversal of the file tree, checking directories as it proceeds. Before the traversal *fsck* makes a series of checks on the root inode, and if it is beyond recovery *fsck* will fail here. Starting at the root, pass 2 performs a set series of checks recursively on all subdirectories. A typical iteration of the recursive traversal is as follows: read the directory's inode, read each data block in the directory, check each entry in each data block. Any entries that point to subdirectories are recursively traversed. Each directory must have one and only one entry for "." and "..", and these entries must contain the correct inode numbers. The states of the inodes pointed to by all the other entries are also checked. If any entry points to an unallocated inode, or an inode with bad or duplicate blocks, the user can choose to remove that entry. These checks ensure that all directory entries are valid. Pass 2 also collects data for future passes. For each directory that is successfully visited, pass 2 will mark it in the state map. Finally, since a directory entry constitutes a link to an inode, pass 2 keeps link counts that will later be compared to the link counts recorded in pass 1. Thus pass 2 compiles a picture of the file system as determined by the contents of the data blocks of all directories that are connected to the root of the file system tree.

Pass 3 finds any directories that have somehow become disconnected from the file system tree. During pass 1 all inodes that identified directories were marked in the state map. Then in pass 2 all directories that were connected to the root by any path were again marked in the same state map. Hence pass 3 needs only to scan the state map for those inodes that were marked as directories in pass 1 and not visited in pass 2. When pass 3 finds such an inode it travels up the parent link (the ".." entry) until it finds a parent inode that is connected to the file system tree. The user is prompted to determine if the sub-tree should be connected as a subdirectory to the "lost+found" directory. It would appear that pass 3 could reconnect the directory to its original parent as pointed to by ".." rather than to the "lost+found" directory, but it does not. If the "lost+found" directory is either missing or full, then the attach attempt fails, and the directories remain disconnected. Once a lost directory is reconnected, pass 3 goes back and executes pass 2 on the reconnected portion of the directory. This reiteration of pass 2 will correct any errors in the connected subdirectory, while marking the new directory as visited in the state map, and correctly updating the link counts.

Pass 4 checks that the link counts to all inodes are correct, and also does cleanup work for the previous passes. For all allocated inodes, pass 4 compares the inode's link count to the link count determined by pass 2. If pass 2 found links to the file, but the link total is different than what the inode claims, the user is prompted to change the link count to the new value. If there are allocated inodes that pass 2 found no links to, the user is given the option of either having the file attached to the "lost+found" directory, or deleted (by clearing the inode). While pass 4 is checking link counts, it also does a cleanup on any bad inodes that remain. Any inodes that refer to directories that were not connected to the file system tree (this can come about due to a failure to connect a directory in pass 3) are cleared. Any inodes that refer to illegal or duplicate blocks (in pass 1 or 1b), and which the user wishes to clear, will also be cleared now. Finally, any inodes that had no links to them and that were not reattached during pass 4 will be cleared. An inode is cleared by marking it and all its associated blocks as free in the appropriate maps. Thus pass 4 guarantees that all inodes are either attached in some way to the file system tree or clear, and that all the internal maps in *fsck* accurately reflect the corrected file system tree.

Finally, *fsck* executes pass 5 which ensures that all the cylinder group information agrees with its own findings. For each cylinder group *fsck* will in turn create a template cylinder group which it will fill with all the correct maps and summary information according to its findings. The inode maps, block maps, and summary statistics are then each checked against the actual cylinder groups. If there is ever a discrepancy, a message is printed to describe what area disagrees (inodes, blocks, or summary statistics), and the user is prompted to have the new

cylinder group information written over the old. Finally, after all the cylinder groups have been checked, pass 5 will check if the super block summary information agrees with the corrected cylinder groups. Again, any discrepancies will be replaced with the new information. *Fsck* is now done checking and correcting the file system. All that remains is for *fsck* to inform the user as to whether or not the file system is now in a state that will allow the machine to enter multi-user mode.[Kowalski 1983]

It must be emphasized, that after a crash, *fsck* does more than just garbage collection. A crash may cause the bit maps to have bits set for inodes or data blocks that are not actually linked into the file system structure, or the maps could just as likely contain bits that are not set for inodes or data blocks that actually are allocated and linked into the file system structure, because the bit maps are updated asynchronously. The former condition is harmless and results in the temporary "loss" of some items that are actually free, while the latter case would result in duplicate allocation, which is the most critical reason why *fsck* must *always* be run during a reboot after a crash, regardless of the cause. Of course, if the system is brought down cleanly, such as with a shutdown command and "umount -a", then all bit maps and other modified data will be correctly written to disk, and no checks will be needed for a reboot. However, crashes are common enough to make such a utility as *fsck* essential for any UNIX system.

2. The *fsck* Program: Inefficiencies & Improvements

This section will describe the problems encountered with *fsck*, and the steps taken to fix them. The *fsck* program is unusual in that it is normally only run at one particular time (when booting the system) and never again as long as the system is up and running correctly. In a typical UNIX system that experiences few crashes the *fsck* program might only be run once a month. However, on experimental machines, the system may frequently be taken down or suffer unexpected crashes. It was on such a machine that *fsck* starting creating something of a problem. This was an Alliant FX/8 with 58M of main memory and 2534M of disk storage. There were 7 physical disks that had been divided into 17 different partitions. The file system on each partition needed to be checked during each boot of the machine. The time to boot the machine was around thirty minutes, of which twenty-one were due to running *fsck*. Having thirty or more users wait an extra twenty minutes for a boot instead of working, when the machine may go down four or more times a week, is an extreme waste of valuable time.

For efficiency reasons, the development of a new version of *fsck* was done on an Alliant FX/1, with 16M of main memory, and 855M of disk memory on 3 physical disks, divided among 8 different partitions. The first step taken in searching for methods of improvement was to insert timing measurements into *fsck* to discover which portions of the program were taking up the most time. The measurements were done using the UNIX times() procedure to measure both system and user CPU time, and using the UNIX gettimeofday() procedure to measure the passage of wall clock time. Each pass of *fsck* was evaluated individually, and the results showed that the first two passes took up the majority of the time (Figure 2.1). In fact, the total of pass 1 and pass 2 averaged about 94% of the time needed to check an individual file system. For this reason efforts were concentrated on making the first two passes faster, while maintaining the functionality of the original program.

When examining the timing results it was evident that pass 2 spent most of its time doing I/O. Examination of the code showed that the recursive tree traversal of pass 2 was causing unnecessary reads. This is because *fsck* has only one buffer for inodes, and only one buffer for data blocks. Every time pass 2 recurses on a subdirectory the current value of those buffers is lost. This means that on every return from a recursion the data block must be re-read, and if there are multiple data blocks in the directory the inode will also have to be re-read to find the number of the next data block. Thus pass 2 generates a minimum of three reads for every directory in the file system tree. All that really needs to be done is to read each data block once,

Real Time Used by Pass 1 and Pass 2 of *fsck*				
File System	Seconds in Pass 1	Seconds in Pass 2	Seconds in *fsck*	%Time in Passes 1 & 2
1	20.41	8.98	31.87	92%
2	61.43	65.82	132.29	96%
3	14.92	7.13	25.51	86%
4	53.89	53.82	116.28	93%
5	69.58	88.65	165.72	95%
6	16.30	3.33	22.30	88%
7	65.94	53.84	125.02	96%
8	59.40	3.49	71.98	88%
Totals	361.87	285.06	690.97	94%

Figure 2.1: Time Used by Passes 1 & 2 in *fsck*

check all its entries in sequence without recursing, write out the block if it was modified, and then go on to the next data block.

To avoid this redundant disk I/O, pass 2 has been completely rewritten, taking care to preserve all the functions of the original. The idea behind the new algorithm for pass 2 is to remove the reads from the tree traversal. From reading the inodes one can gather the block numbers for all directory data blocks. These data blocks can then be read in a step by step process. A single step of the process is to read and check all directory data blocks for an inode, and this step is repeated for each directory inode. There still needs to be some sort of tree traversal to mark connected directories, and to check the ".." links, but the new pass 2 is written so that this traversal can be done totally in memory after all the I/O has been done.

First, all the reading of inodes is removed from pass 2. Pass 1 already reads all the inodes, so a new map called a directory map is added, and pass 1 collects the data block numbers from all directory inodes and stores them in this map for later use in pass 2. To check all the data blocks, the original plan was to traverse the directory map in sequence, reading and checking all data blocks for an inode, and then moving on. However, since most directories use only one data block it turns out to be more efficient to quicksort the inodes based on the block number of the first data block for each inode. This reduces the amount of disk arm movement and seek time. The old pass 2 checks are performed on each directory data block, except for the checking of the ".." link, since the parent of any given directory is still unknown at this time.

As the data blocks are being checked, the inode numbers for all subdirectory entries are stored in a linked list under their parent directory in the directory map. After all the directory data blocks have been checked, the directory map contains all the information needed to do a quick traversal of the tree to mark directories as visited, and to check the validity of ".." links. An additional bonus of this new pass 2 is that it is no longer necessary for pass 3 to go back and check the data blocks for any directories it might reconnect. This is because pass 2 now checks all directory data blocks, instead of just checking those that are attached to the root.

When the timing tests were repeated for the new pass 2, they consistently showed that 1/3 of the time used by the old pass 2 was system time (Figure 2.2). The code for the new pass 2 also proved to be more efficient in terms user time, although the amount of improvement was only 5% to 25% less user time.

Time Used by Pass 2						
	Old *fsck*			New *fsck*		
File System	Seconds in Pass 2	User Ticks	System Ticks	Seconds in Pass 2	User Ticks	System Ticks
1	8.98	44	118	2.86	35	40
2	65.82	250	815	18.43	203	279
3	7.13	38	94	2.62	31	33
4	53.82	244	691	16.16	196	240
5	88.65	434	1121	26.21	324	394
6	3.33	22	42	1.62	21	16
7	53.84	210	706	15.27	169	242
8	3.49	13	37	1.90	24	14

Comparison of Old and New Pass 2 Time Usage				File System Statistics		
File System	Percentage Fewer Seconds	Percentage Fewer User Ticks	Percentage Fewer System Ticks	Number of Directories	Number of Directory Blocks	Total Number of Inodes
1	68%	20%	66%	87	88	16384
2	72%	19%	66%	621	622	47104
3	63%	18%	65%	73	73	12288
4	70%	20%	65%	522	530	45056
5	70%	25%	65%	863	878	53248
6	51%	5%	62%	32	34	12288
7	72%	20%	66%	534	537	45056
8	46%	-85%	62%	29	29	53248

Figure 2.2: Time Used by Pass 2 in Old *fsck* vs. New *fsck*

There was one special case, where file system 8 actually took more user time to check with pass 2. However, as indicated by Figure 2.2, this is a case where the file system has a large number of inodes, of which only a very few are used for directories. The old pass 2 found directories by traversing the file system tree, while the new pass 2 has to search through all inodes in the state map to find the directories. This search does not take long, but it becomes noticeable when there are less than 30 directories among greater than 53,000 inodes. However, such empty file systems are a rare case, and in terms of real time the new pass 2 showed itself to take about 64% less time than the old pass 2. Although not shown in the figures, about 5% of the overall time savings resulted from sorting the directories by the first block number before reading.

After improving pass 2 as much as possible, attention was turned to pass 1 which in a number of cases took even longer than the old pass 2. Again timing results were studied, and they showed that pass 1, like pass 2, spent most of its time satisfying I/O requests. However, the code showed that unlike pass 2, pass 1 made no unnecessary I/O requests. Pass 1 simply has to read every inode in the file system it is checking. Yet, when closely examined, it appeared that pass 1 was not taking proper advantage of the file system structure.

All the inodes in a each cylinder group are stored sequentially on disk. When reading inodes, pass 1 should take advantage of the fact that it is doing sequential reads, but it does not. Pass 1 reads inodes into a buffer that is the size of one file system data block. On the test machine's UNIX file system, each cylinder group is allowed a maximum of 2048 inodes, and a data block is 4096 bytes long and capable of holding 32 inodes. By reading the inodes in such a small buffer, pass 1 wastes time in two ways. First there is the extra operating system overhead involved in doing a read. The overhead time may not be significant for a single read, but pass 1 is repeating that overhead 64 times, when it could potentially read all 2048 inodes with one call.

The second waste of time is less obvious, and depends on the particulars of the disk system. On any disk there is a latency time while the disk waits for the proper data block to spin under the disk head. With the old pass 1, a data block was read, the inodes within were checked, and then a second data block was read. While the inodes are being checked the disk keeps spinning, and when the next read request arrives there will be another wait for the proper block to be aligned. However, if multiple blocks are requested at once, the disk controlling software should be able to read all the requested blocks in sequence, after only waiting for the first block to pass under the read head. Thus, ideally, pass 1 should read 2048 inodes into a buffer, and then process them. This way, the overhead of a read call and the disk latency time are only experienced once for each cylinder group, instead of 64 times.

Before installing the ideal case of reading 2048 inodes, one must consider the ramifications. For inodes of 128 bytes each, pass 1 would need a buffer of 256K bytes. That is a lot of extra memory to add to a program that is already memory intensive. For this reason, tests were performed to see how the time for completion of pass 1 would improve as buffer size was increased. Timing measurements were done on pass 1 for a number of different buffer sizes. The tests used 4K, 8K, 16K, 32K, 64K, 128K, and 256K buffers. The results of CPU time vs. buffer size for a typical file system are shown graphed in Figure 2.3. The dashed line shows user time, and the solid line shows system time. The other file systems show some variation among the maximum and minimum times, but all of the graphs have the same shape and ratios of improvement.

In the graph, the dashed line levels off quickly, and the only reason it decreases at all is because there are fewer procedure calls to the *fsck* read routine, which in turn calls the system read routine. The solid line decreases much more dramatically, but again levels off for the larger buffer sizes. The reason behind the quick leveling off is apparent. Because of the limit of 2048 inodes for each cylinder group, a buffer that can hold all 2048 will give the best time possible. For each doubling of buffer size, the time from the previous buffer size to the best possible time is halved. In order to get significant improvement without wasting too much memory, a 64K buffer (which can hold 512 inodes) was adopted for the new pass 1. With this larger buffer the new pass 1 finishes in slightly less than 1/2 the time of the old pass 1. The decrease in real time is only partially accounted for by the decrease in CPU time (Figure 2.4), which would imply that there has also been a significant savings due to less disk latency time.

Both the changes in pass 1 and the changes in pass 2 were made at the expense of greater use of main memory. The new pass 1 uses a much larger inode buffer, and the new pass 2 uses a new directory map structure. Therefore one must determine if this increased memory use will adversely affect the performance of the new *fsck* program. In particular it would be bad if the extra memory used caused paging out to disk, since the goal was to decrease disk activity.

It has been stated that *fsck* is executed when it has no competition for CPU time. While this was true for the timing tests, it is not always true when *fsck* is run during a system boot. When *fsck* is checking the file systems at boot time there is usually one copy of *fsck* running for each disk attached to the machine. Thus the test machine has three *fsck* processes running in parallel at boot time. If memory use of the new *fsck* was to increase greatly it could cause page-outs as the three processes compete for main memory. However, when tested it was found

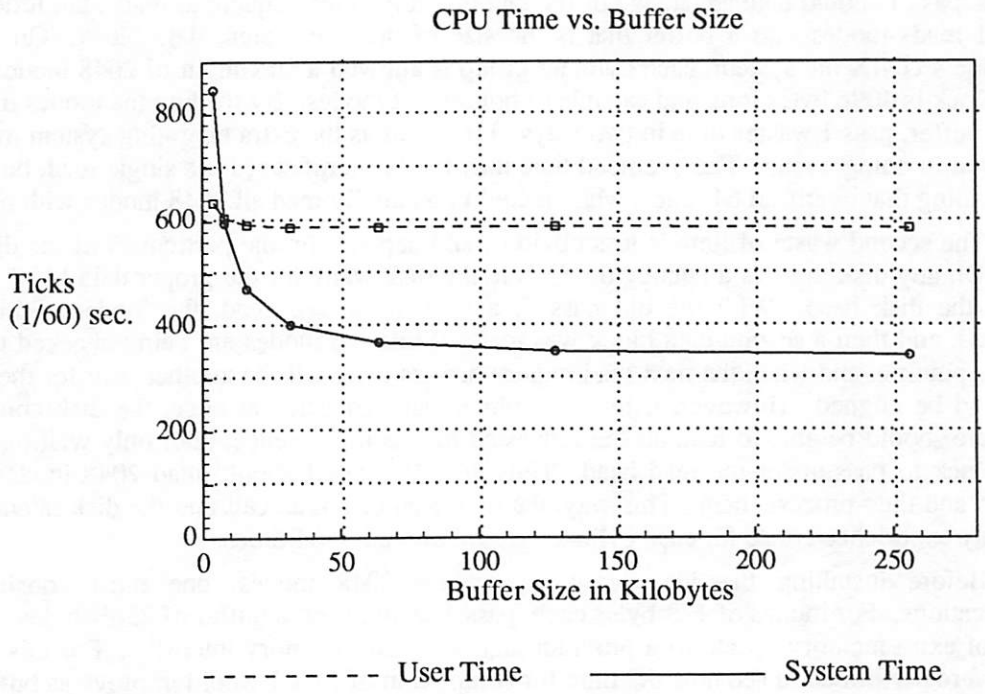


Figure 2.3: Graph of CPU Time vs. Buffer Size for Pass 1

that the old *fsck* used approximately 4.4MB of memory, while the new *fsck* used approximately 4.9MB of memory. This is an increase of slightly more than 10%, and caused no problems since the test machine has 16MB of main memory available. Since the additional memory usage is small, it is likely that any machine which had enough memory for the old *fsck* to run properly will have no additional difficulties with the new one.

Much of the inefficiency located in the old *fsck* has been identified, and steps taken to reduce it have been detailed. Likewise, a description has been given of how the new code manages to perform all the functions of the old. Finally, the reductions in both CPU and real time have been documented at each step. Later, in section 5, performance times are given for rebooting with either the old or new *fsck* installed.

Time Used by Pass 1						
	Old *fsck* (4K Buffer)			New *fsck* (64K Buffer)		
File System	Seconds in Pass 1	User Ticks	System Ticks	Seconds in Pass 1	User Ticks	System Ticks
1	20.41	206	298	11.71	184	131
2	61.43	673	855	36.14	602	370
3	14.92	141	191	6.20	124	67
4	53.89	566	687	21.61	511	225
5	69.58	862	862	31.21	762	295
6	16.30	146	205	7.41	130	81
7	65.94	690	805	32.17	618	342
8	59.40	520	769	19.70	449	223

Comparison of Old and New Pass 1 Time Usage			
File System	Percentage Fewer Seconds	Percentage Fewer User Ticks	Percentage Fewer System Ticks
1	43%	11%	56%
2	41%	11%	57%
3	58%	12%	65%
4	60%	10%	67%
5	55%	12%	66%
6	55%	11%	60%
7	51%	10%	58%
8	67%	14%	71%

Figure 2.4: Time Used by Pass 1 in Old *fsck* and New *fsck* with 64K Inode Buffer

3. The New *fsck* Program: Verifying Functionality

Before the new *fsck* could be installed in the larger machine it had been designed for, it was tested to make sure it still performed all of the functions of the old version. It has already been shown that all of the information is available for the new program to make the same checks, but the code needed to be tested for each of the possible error conditions before it could be trusted enough to be installed. There is not space here to describe in detail all the tests performed, so just the basic test strategy will be covered.

Portions of an existing file system were selectively corrupted to generate most of the error cases checked for in the original *fsck* program. Both the new and old versions of *fsck* were used to fix the errors, and the results were compared to guarantee that both versions fixed the problems identically. Although only passes 1 and 2 of the original program had been modified, all the passes were tested.

The only discrepancy of note occurred in the new pass 2. When the original *fsck* found an error in a directory block it would inform the user of the pathname of the offending directory. Since the new pass 2 does not do a tree traversal to check directory blocks, it is unaware of the pathname at the time the error is detected. To correct this, a function was added to find the pathname of a directory by traversing the ".." links up to the root. This allows the new pass 2 to still provide the same output as the old. The search for the pathname does provide some extra overhead when correcting errors, but on a typical execution of *fsck* the number of corrupt directories should be very small compared to the total number of directories being checked. Overall the new *fsck* provides identical functionality to the old, with almost identical output, while allowing faster boot times than the original program.

4. The New *fsck* on a Vax 785 running UNIX 4.3BSD

In an effort to show that the modifications made to *fsck* were applicable to any machine that uses BSD *fsck*, the new *fsck* was modified to run on a Vax running BSD version 4.3 of UNIX. This section will list some of the difficulties encountered in transporting *fsck* to the Vax.

The version of *fsck* distributed with BSD version 4.3 of UNIX differs in several ways from the *fsck* in BSD version 4.2. However, most of these differences involve checks for additional types of file system corruption, and all of the functions of the 4.2BSD *fsck* were still in the 4.3BSD *fsck*. Likewise the 4.3BSD version of *fsck* still had all the inefficiencies involving unnecessary I/O that had been found in the 4.2BSD version. Thus the 4.3BSD version of *fsck* should derive the same benefits as the old version when the changes described previously are added.

However, one difference that affected these changes was that the 4.3BSD version of *fsck* allows the user to attempt to allocate a new root or "lost+found" directory if the old one is corrupt or missing. To accommodate this change, the new directory map was modified to contain the inode and block numbers for these newly created directories. Also, the 4.3BSD version of *fsck* allows the user to expand the "lost+found" directory if it is too full to add another entry. This expansion means that the block list for the "lost+found" directory in the directory map has to be adjusted to include the new block. Other than these adjustments, all the improvements for *fsck* that were developed on 4.2BSD UNIX were easily moved to 4.3BSD UNIX.

There was one noticeable difference between the Alliant running 4.2BSD and the Vax running 4.3BSD that showed up when doing timing tests of the new *fsck*. This difference was that the increased buffer size that was added to pass 1 did not seem to benefit the 4.3BSD machine as it did the 4.2BSD. A graph of the effect of increased buffer size on a typical file system on the 4.3BSD machine is given in Figure 4.1, and it clearly shows that increased buffer size gives almost no improvement in execution time. However, Figure 4.2 is a similar graph which shows that there has been some improvement in real time. The disks used on both machines have very similar functional specifications, so the difference appears to be in the controlling software. It may be that the disk controlling software on the 4.3BSD machine does not allow the reading of multiple pages in a single disk read system call, or conversely it may always be reading a full track. We did not study the details of the 4.2BSD and 4.3BSD kernels, or the disk drivers for the Alliant and Vax, to determine why the Alliant showed the kind of improvement we expected, while the Vax did not. However, the improvement in real time shows that the Vax still gains some benefit in the form of less latency time.

Despite the fact that differences in disk software may somewhat affect the degree of improvement, the ease of transporting the changes from the Alliant(4.2BSD) to the Vax(4.3BSD) shows that these changes can fairly easily be applied by anyone who uses the BSD *fsck* program.

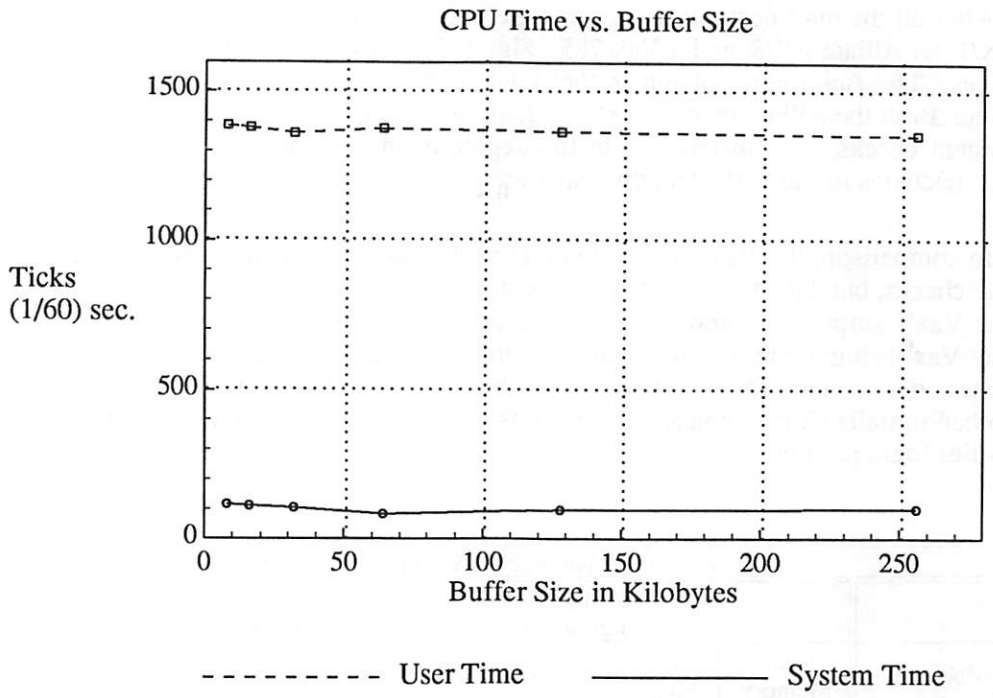


Figure 4.1: Graph of CPU Time vs. Buffer Size for Pass 1

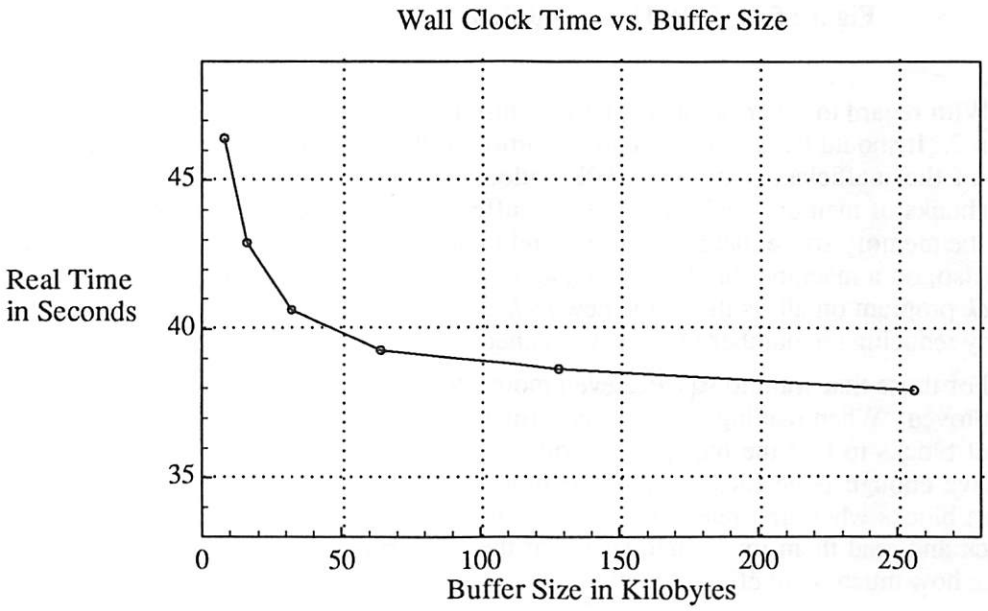


Figure 4.2: Graph of Real Time vs. Buffer Size for Pass 1

5. Overall Timing Results and Conclusions

After all the modifications were complete, the new *fsck* program was installed on an Alliant FX/1, an Alliant FX/8, and a Vax 785. Figure 5.1 shows the relevant statistics on the three machines. The *fsck* times column is the time to check all file systems during a boot of the machine. Both the Alliant machines showed approximately the same degree of improvement in file system checks. The differences in the degree of improvement in total boot times on the Alliant machines is due to the lengthy hardware configuration procedures necessary for the Alliants.

In comparison, the Vax did not do quite as well as the Alliants in decreasing the time for full file checks, but did show better than expected improvement considering its problems in pass 1. The Vax's surprisingly good performance on pass 2 is probably a result of the file systems on the Vax being more complex, and having more directories than those on the Alliant machines. The column for boot time shows the total boot time including both file checks and any other initialization overhead, meaning from the time one hits the "boot button" to the multi-user login prompt.

Fsck Performance on Various Machines								
	Machine Configuration				*Fsck* Times (Min:Sec)		Boot Times (Min:Sec)	
Machine	Main Memory	Disk Memory	Disks	Partitions	Old	New	Old	New
Alliant FX/1	16M	855M	3	8	10:00	3:42	14:35	8:06
Alliant FX/8	58M	2534M	7	17	21:30	7:46	29:35	15:51
Vax 785	16M	1435M	4	16	14:21	6:47	16:40	9:06

Figure 5.1: *Fsck* Times and Boot Times for Various Machines

With regard to other problems, let us return briefly to the extra memory use pointed out in section 2. It should be noted that a good portion of the extra memory used in the new *fsck* is a result of the inefficient way the UNIX malloc() routine allocates memory. When allocating large chunks of memory such as the inode buffer and the directory map, malloc() will allocate twice the memory space that is asked for, and then fail to ever break up the extra space for later use. Also, on a machine that has just enough main memory to fit the parallel execution of the old *fsck* program on all its disks, the new *fsck* could still be run without causing page-out problems by reducing the number of file system checks that are run in parallel.

For those that want to squeeze even more speed out of *fsck*, it may be that pass 1 can still be improved. When reading inodes to construct the block map in pass 1, *fsck* must also read all indirect blocks to find the blocks they point to and mark them in the block map. On systems that have enough large files, it could be more efficient to store the block numbers of all the indirect blocks when first reading the inodes in pass 1, and then to sort the indirect blocks, and go back and read them in a single sweep of the disk arm. We did not find the time to try this and see how much of an effect it might have.

Another difficulty with *fsck* is a lack of information given at user prompts in the early passes. In pass 1 *fsck* asks to remove inodes while only giving the user the inode number of the file to be deleted. Most users cannot identify the actual file from the inode number, and may discover later that they deleted a file they would have preferred to save. It might be better to flag bad inodes in pass 1, and then collect the names of the flagged files during pass 2, so the user can then make informed decisions about which files to delete.

In conclusion, the modifications to *fsck* as described in this paper have proven to decrease the boot time by almost 50% on several machines, thus providing a valuable savings in computer time to the users. Since these changes have proven to be effective on different UNIX machines, and have also proven to perform all the same functions as the old *fsck* program, they should be of considerable benefit to any UNIX machine that currently makes use of the BSD *fsck* file system consistency checking program.

References

Bach, M. J. **The Design of the UNIX Operating System.** Prentice-Hall Inc., Englewood Cliffs, New Jersey 07632 (1986).

Kowalski, T. J. *Fsck - The UNIX File System Check Program.* **Computer Systems Research Group, Berkeley.** (July, 1983).

McKusick, K. M., Joy, W. N., Leffler, S. J., and Fabry, R. S. *A Fast File System for UNIX.* **ACM Trans. on Computer Systems.** vol. 2, (Aug., 1984) pp. 181-197.

Lessons of the New Oxford English Dictionary Project

Tim Bray

Centre for the New OED
University of Waterloo
Waterloo, Ontario

ABSTRACT

The *Oxford English Dictionary* is a cultural monument and one of the great achievements of human scholarship. The computerization efforts at the University of Waterloo are just another chapter in this 130-year continuing effort.

The work at Waterloo has resulted in several important achievements, including a general purpose system for transduction of text based on regular grammars, a flexible system for text search that may be the fastest existing today, an entirely new model for text databases, and an original user interface architecture. Implementations of these ideas are now in use at several sites, and are playing an important role in the continuing work on the *OED*.

Also, we have learned some valuable lessons about text management, user interfaces, and in particular the application of UNIX[1] to these problems.

1. History

Discussions on the creation of a ''New English Dictionary'' began in the Philological Society in the 1850s. After many false starts, an editor was appointed and the first volume published in 1884. Fourty-four years later, with the addition of three other editors and the death of two, the first version of the *OED* [Murray] was complete. A four-volume supplement was produced between 1959 and 1986. This work, forever unfinished, continues at Oxford and at Waterloo, Ontario, and UNIX is central to it.

The *OED* is a masterwork of comprehensiveness and scholarly consistency. It attempts to provide complete coverage of every sense of every word that has appeared in a published work in English since the year 1150. Each is supported by a list of quotations from the literature exhibiting its history. Table 1 presents some statistics about the *OED*'s contents.

1. UNIX is a trademark of AT&T.

Table 1. The Contents of the *OED*

	OED	Supplement	Total
Pages	15,487	5,601	21,088
Main Entries	252,259	69,372	321,631
Subentries	98,555	44,103	142,658
Quotations	1,861,212	560,415	2,421,627
Cross References	474,582	99,467	574,049
Megabytes	401	129	530

However, the dictionary has suffered from the fact that unlike the language, it is immutable — literally "cast in metal". It and its supplements are not completely up-to-date on the day they are published, and become more out of date each year.

The New *OED* project at Oxford University Press (OUP) and the University of Waterloo is a response to that problem. OUP is concerned with publishing the *OED* Second Edition and selling it in worldwide markets. Waterloo is concerned with building a database architecture and delivery system suitable for the *OED* and other such texts. While the *OED* is the first and most visible application, we believe that it is just one example of an important class of large, highly structured bodies of text. Furthermore, we believe that the state of the art in the management of such text lags behind other areas of computer science in both theory and practice.

2. Technical Achievements

2.1. The Nature of Structured Text

The *OED* is typical of highly structured texts. Each entry is composed of a large, complex, hierarchy of morphological information, sense definitions, and supporting quotations (see figure 1). This structure is regular and consistent in general, but highly irregular at the detail level. Components are often missing or appear in non-standard order; they are wildly variable in size, with entries ranging from about 100 bytes to over 500,000. This complexity and variability of structure encodes information just as important as that specified explicitly in the text.

To complicate things, a small proportion of irregular constructs are simply errors, which may have been introduced by the original authors, during the initial typesetting of the work, or during the data acquisition. This is a problem because the database is so large that even a very small error rate leads to a very large number of errors — too large to consider fixing by hand in less than a few years.

2.2. Text Transduction

The first problem faced by the New *OED* group at OUP was capturing an on-line version of their data. Optical scanning proving impracticable, it was typed in by hand; a 180 person-year effort! However, this resulted in a form of the data that contained principally typographical markup and was not suitable for any serious on-line use. The goal was to transform it into a descriptively-tagged version in the SGML [ISO] style.

Jacinth (dʒæ·sinþ, dʒē·sinþ). Forms: 3-7 iacinot(e, 4 iacynkt(e, -synkt, -cintt, 4-6 iacynct, iacynt(e, 4-7 iacynth, 6 iassink, 6-7 iacinth(e, iacint, (7-8 jacent, -int), 7- jacinth. See also HYACINTH, and JACOUNCE. [ME. *iacynt*, *iacinct*, a. OF. *iacinte* or late L. *iacint(h)us*, *-inctus*, an alteration of *hiacint(h)us*, L. *hyacinthus*, a. Gr. ὑάκινθος HYACINTH; the *h* being lost and the initial *i* made consonantal; cf. mod.F. *jacinthe*, Pr. *jiacint*, Sp. *jacinto*, It. *giacinto* and *iacinto*.]

1. a. Among the ancients, a gem of a blue colour, prob. sapphire. **b.** In mod. use, a reddish-orange gem, a variety of zircon; also applied to varieties of topaz and garnet. (= HYACINTH 1.)

c 1230 *Hali Meid.* 43, & tah is betere a briht iacinct þen a charbucle won. 1382 WYCLIF *Song Sol.* v. 14 Goldene, and ful of iacynctis. 1535 COVERDALE *Ezek.* xxviii. 13 Deckte with all maner of precious stones, with Ruby, Topas, Christall, Iacynte. 1555 EDEN *Decades* 236 Iacinthes growe in the Iland of Zerlam. They are tender stones and yelowe. 1567 MAPLET *Gr. Forest* 11 The Iacinct is blew, and of nigh neighborhoode with the Saphire. 1630 DRAYTON *Muses' Elys.* x. (R.), The yellow jacinth, .. Of which who hath the keeping, No thunder hurts nor pestilence. 1762–71 H. WALPOLE *Vertue's Anecd. Paint.* (1786) I. 154 The dagger, in her grace's collection, is set with jacynths. 1861 C. W. KING *Ant. Gems* (1866) 22 The greater part .. of what are now termed Jacinths are only Cinnamon Stones of a reddish kind of Garnet.

† c. (In Wyclif's Bible, rendering L. *hyacinthus*): A dyed fabric of a blue or purple colour. *Obs.*

1382 WYCLIF *Exod.* xxv. 4 Iasynkt that is silk of violet blew. *Ibid.* xxviii. 15 The breest broche .. thou shalt make with werk of dyuerse colours, after the weuyng of the coope, of gold, iacynkt [1388 iacynt], and purpur.

d. The colour of the gem (see b above); in *Her.* name for the colour *tenné*, in blazoning by precious stones (= HYACINTH 1 c).

1572 J. JONES *Bathes Buckstone* 11 b, If it [the urine] be higher, then ambre or betwene it and iacincte, yellowish or chollerique red. 1572 BOSSEWELL *Armorie* II. 66 The fielde is of the Iacinthe. 1688 R. HOLME *Armoury* I. ii. 12/2.

† 2. A plant; = HYACINTH 2 (a and b). *Obs.*

[1398 TREVISA *Barth. De P. R.* XVI. liii, An herbe of þe same name is liche þerto [the stone Iacinctus] in coloure.] 1567 MAPLET *Gr. Forest* 47 Iacinct is an Herbe hauing a purple flowre. 1597 GERARDE *Herbal* I. lxxvii. (1633) 106 The white-floured starry Iacinth. 1629 PARKINSON *Paradisi* xi. 122 Our English Iacinth or Hares-bels is so common everywhere. 1727 *Philip Quarll* 244 Junquils, Tuberoses, Jacents, and other delightful Flowers. 1760 J. LEE *Introd. Bot.* App. 315 Jacinth, *Hyacinthus*.

3. *attrib.* and *Comb.* (in senses 1 and 2).

1526 TINDALE *Rev.* ix. 17 Havynge fyry habbergions of a Iacynct coloure. *a* 1585 SIDNEY *Arcadia* I. Wks. 1725 I. 20 Her forehead Jacinth-like, her cheeks of Opal hue. *Ibid.* 107 The excellently fair queen Helen, whose jacinth-hair curled by nature .. had a rope of fair pearl. 1591 PERCIVALL *Sp. Dict.*, *Iacinto*, a iacint stone, a iacint flower. 1811 PINKERTON *Petral.* II. 129 Consisting of quartz and of jacint, so that it may be called jacint rock. 1842 TENNYSON *Morte d'A.* 57 Myriads of topaz-lights, and jacinth-work.

Figure 1. An *OED* Entry

A variety of methods were explored, including programs written in *lex/yacc*, prolog, and PL/I macros. Most quickly ran into trouble with the immense size and complexity of the grammars necessary to describe the required input and output structures. The eventual solution was found in the form of INR and Lsim, programs developed at Waterloo [Johnson], [Kazman]. INR reads a grammar for an extended regular language and generates the corresponding minimized N-tape deterministic finite automaton. If the grammar describes a 2-tape automaton, it can be used as a transducer by treating the input as tape 0, and placing on the output stream the corresponding sequence of symbols that would appear on tape 1. This function is performed by the lsim program.

It is important to realize that this only made the job possible, not easy. It was necessary to reverse-engineer the dictionary to ascertain its deep structure, which had never been written down.

INR and lsim have continued to find valuable use as general-purpose nonprocedural tools for transforming text from one form to another, a problem which arises repeatedly in the area of text management. They have also taught us many valuable lessons about the use of grammars to model and process structured text.

2.3. The PAT Text Search System

```
-r--r--r--  1 ghgonnet 400989181 May 21 08:39 OED
-r--r--r--  2 tbray    128709256 May 27 08:45 S
```

Figure 2. The text of the *OED* and Supplement

Once we had the *OED* nicely formatted in disk files (see Figure 2) it became painfully obvious that we needed new tools to process it. Not only does it take 45 CPU minutes on a Sun 3/160 just to *getchar()* through the OED, many of the UNIX utilities that we all know and love obstinately refused to deal with even "small" pieces of this text, for the irrelevant reason that it contains no newlines (more on this in section 3).

The first major step forward was the development of PAT, a tool for rapid searching of large text files. PAT, developed on UNIX but very portable, is a combination of some old ideas, preprocessing and Patricia trees, with some new ideas, including "semi-infinite strings", "docs", and arbitrary index points. Figure 3 is a record of a short PAT session.

As appropriate for a UNIX tool, PAT also has a silent mode, suitable for use by other programs. However, this mode departs from standard practice in that it does not produce a text stream as output from a query. Rather, it produces a list of byte offsets into the database which may be used to retrieve arbitrary portions of match and context. This is necessitated by the great size and recordlessness of our databases.

2.3.1. Preprocessing

PAT searches text very rapidly by using an index based on the Patricia tree structure built in a preprocessing pass [Knuth]. This index has the property that searching is logarithmic in the size of the database and independent of the size of the answer set [Gonnet88]. PAT is amazingly fast — on a Sun 3/160, it can find all the matches for an

```
watsol% pat /usr/newoed/OED
    University of Waterloo
Pat 3.3 (Jan/88) Text searching system
>> computer
  1: 10 matches
>> docs Q including computer
  2: 6 matches
>> pr.docs.Q
 50925396, ..<Q><D>1646</D> <A>Sir T. Browne</A> <W>Pseud.
Ep.</W> <sc>vi. </sc>vi. 289 <T>The Calenders of these com-
puters. </T></Q>..
 50925521, ..<Q><D>1704</D> <A>Swift</A> <W>T. Tub</W> vii,
<T>A very skilful computer. </T></Q>..
393399529, ..<Q><D>1727</D> <A>Swift</A> <W>Let. to very
young Lady</W> Misc. II. 337, <T>I think you ought to be
well informed how much your Husband's Revenue amounts to,
and be so good a Computer as to keep within it. </T></Q>..
373590780, ..<Q><D>1750</D> <A>Johnson</A> <W>Rambler</W>
No. 71 11 <T>The computer..believes that he is marked out to
reach the utmost verge of human existence. </T></Q>..
354344933, ..<Q><D>1846</D> <A>G. J. Dow (</A><W>title</W>)
<T>Calculus, the turfite's computer. </T></Q>..
 50925722, ..<Q><D>1855</D> <A>Brewster</A> <W>Newton</W>
II. xviii. 162 <T>To pay the expenses of a computer for
reducing his observations.</T></Q>..
>> quit
used 0.88 cpu seconds
```

Figure 3. A Simple PAT Session

arbitrary string in the 400-Mb *OED* in less than 1 second elapsed. In effect, it turns the database text into a content-addressable store.

This speed does not come for free, of course. The price is paid in the preprocessing, which is very time-consuming. A straightforward analysis shows that the best algorithm is `O(n log n)`, but this turns out to be totally useless, as it involves `O(n log n)` random disk accesses. Assuming a typical UNIX maximum of about 30 random file accesses per second and given that the *OED* has about 80 million index points, we calculate an elapsed time of some months. Algorithms that are less efficient than `O(n log n)`, but which process large disk files sequentially rather than randomly, do much better. However, there is still much room for improvement.

2.3.2. Semi-Infinite Strings

''Semi-infinite strings'' are substrings of the database which extend arbitrarily far to the right from some starting point. There is an interesting Abstract Data Type for semi-infinite strings, with an accompanying algebra. The practical import is that PAT indexes the semi-infinite string starting at each index point. This means that one finds all the matches for ''to'', ''to '' (note space), ''to be o'', and ''to be or not to be'' with equal speed.

UNIX users should feel comfortable with this model of data, which is reminiscent of UNIX byte-stream practice.

2.3.3. Docs

Docs (for documents) is a generalization of the ''field'' concept. Any arbitrary set of substrings of the database can be declared a docs and referred to by name. Typically, in an SGML-style tagged text, there would be named docs for each tag of structural interest. Docs can be precomputed and stored in files for efficiency, or can be created dynamically at runtime, using the results of previous searches to restrict the scope of later searches.

To give a concrete example from the *OED*, it would be possible to find all the 13th-century dates, find the quotations containing them (using the docs for quotations), and treat this set of quotations as a docs to allow searching for some other interesting pattern only within 13th-century quotations.

2.3.4. Index Points

When building an index for the *OED*, the first and most important question is what to index. Indexing every character of the text is expensive and counter-productive for the majority of search applications. People usually want to index ''words'', but it is far from clear how to define a word. PAT addresses this problem with a specification matrix, a set of character pairs which define the index points; metacharacters are available to specify things such as ''index a hyphen if it follows an alphabetic character.''

2.4. The Grammar-Defined Model for Text Databases

Early on in the New *OED* project, it became clear that there were serious problems in the direct application of traditional database concepts, such as the relational model, to the *OED* in particular and to structured text in general.

For this reason, we have developed an entirely new model for text databases, in which the schemas are grammars and the instances are parsed strings or ''p-strings''. There is a suite of operators defined on p-strings; such a database is termed a ''grammar-defined database'' (GDDB) [Gonnet87].

A p-string can be thought of as a triple containing:

1. The underlying text from the database,
2. The grammar by which it is to be parsed, and

3. The actual parse tree for the text.

Operators are available to convert back and forth between uninterpreted text and p-strings. Other operators which modify the grammar may be used to produce different views of the database with immense flexibility. Operators which extract and manipulate the parse tree data structure may be used to create new p-strings from old.

We have shown that there is a large and interesting class of queries which cannot even be asked using conventional database access models (for example using SQL), but which are expressed easily in the grammar-defined model.

Our implementation of this model is embedded in a very high level language which also includes a set of conventional string processing primitives. We were forced to use an object-oriented approach by the realization that while a p-string can potentially contain a lot of information about its structure, text, and grammar, many queries can be answered using much less information. For the (very common) queries of the form "does p-string X contain another p-string with root B?", all the information that is needed is, for each p-string, the root nonterminal and a pair of pointers indicating its extent in the database. Accordingly, we implement p-strings as active data structures which start out with only minimal information, but which compute and remember other elements of the p-string structure on demand.

2.5. User Interface

The central problem we face in our user interface work is complexity. The dictionary, and in general any interesting structured text, tends to be complex and difficult to understand. The tools that access it, then, need to be flexible and powerful — hence also often difficult to learn. The problem is how to make these necessarily complex tools easily accessible, and secondly how to present the complex data in a way that aids in its comprehension and editing.

In this area, it is important not to underestimate the power and sophistication of existing technology, represented by a desktop scattered with reference works, offprints, and scribbled notes. This technology has enjoyed some five thousand years of evolutionary development; it is a rich and efficient work environment which provides a very high bandwidth from the desktop to the mind.

We have investigated a variety of solutions to the complexity problems; currently, we find that for representing highly complex text structures, it is hard to improve on the traditional method, namely typography. Figure 4 shows several windows on a workstation displaying the same dictionary entry in a fashion that illustrates the power of combining traditional typography and simple window technology.

We have also investigated hypertext, and at this time remain unconvinced that it is applicable to this type of data [Raymond]. Two major problems are worthy of brief mention:

1. For an existing text, it will be necessary first to fragment it appropriately, then to compute the appropriate links between the fragments, and finally to construct and store the links. All of these steps are difficult and expensive. Likely they will require a much fuller understanding of the semantics of the text than is now achievable.

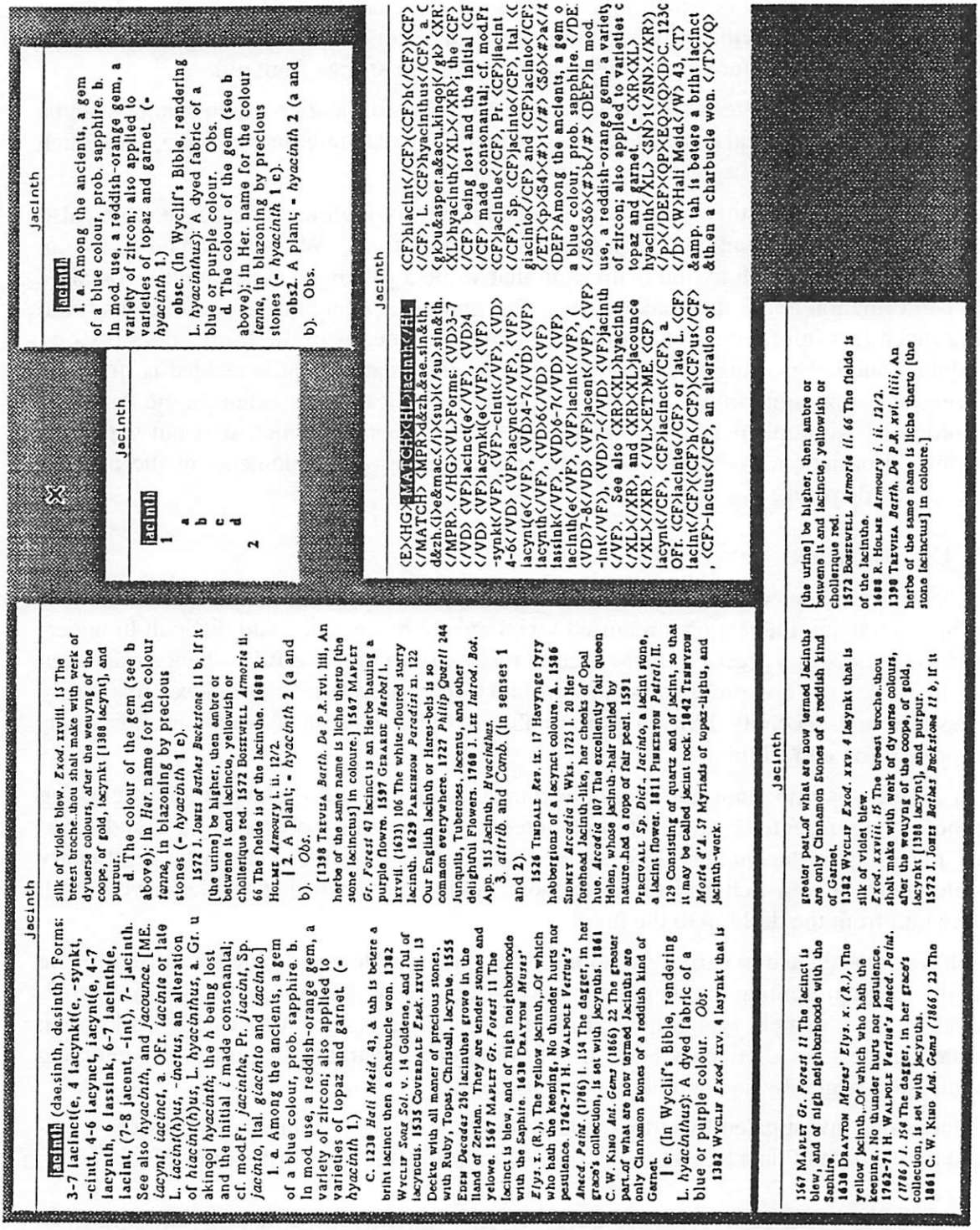


Figure 4. Ways of Looking at Jacinth

2. The *OED* (as with many other scholarly works) was not designed as a set of interconnected bite-sized atoms, but in carefully argued and ordered bodies of text. Because of this, even assuming it were possible to store such documents in a properly linked-up hypertext, it is not clear that this would provide more useful access than our existing tools.

Our user interface is consistent with to the UNIX paradigm in that it is a set of interconnectable tools, rather than a large integrated attempt to solve all possible problems. The tools run in separate processes and exchange data as streams of text with embedded descriptive markup. One important finding is that this technique can provide acceptable performance; a rare worst-case scenario in which one tool must transmit hundreds of kilobytes of text to another causes no more than a few seconds of delay. Now to a certain extent this is a function of the host system, but nonetheless it is good news; an unstructured self-descriptive byte stream is clearly optimal in terms of flexibility and generality.

A subsidiary problem, not unique to our project, is portability. Since the user interface needs modern high-resolution displays and window architecture to be effective, the question is: which window system? It is difficult, at this time, to choose a window system without eliminating a large segment of the potential audience for any software. Our approach has been to use a carefully restricted subset of the X-windows system and to abstract the I/O to an extent where we are hopeful of achieving portability to other window systems.

2.6. Applications

There have been a variety of successful applications of all these techniques, but the most interesting is the *New Shorter OED* (*NSOED*) project. This project is using all our tools in an integrated fashion to support the task of creating a draft version of the *NSOED*. The *OED* text is loaded into a grammar-defined database and processed using the GDDB operators, extracting those entries that are appropriate for the *NSOED*, and performing drastic reorganization of those entries so that they are consistent with the *NSOED* structure. Figure 5 shows a simplified version of the module that decides whether a given *OED* entry is suitable for inclusion in the *NSOED*. The output of this is a set of skeleton *NSOED* entries, ready for processing by the lexicographers, who have been thereby promoted from entry drafters to entry editors. It is our opinion that this program is unique; quite likely, it simply could not have been written using conventional programming methods.

The transduction tools are used to transform the output data to conform to OUP's tagging standards. PAT is used to support *NSOED* research by searching the *OED*, the portion of the *NSOED* that has been completed, and a variety of other research material. The user interface tools will be used to support interactive browsing of this research material.

```
InShorter := proc(e)
  everydate := every qdate in e;
  if size(everydate) = 0
  then false
  else
    dates := year mapped onto (string mapped onto everydate);
    if max(dates) >= 1700
    then true
    elif size(every quote in e where SQuote()) > 0
    then true
    else false
    fi;
  fi;
end;

SQuote := proc(q)
  author := string(auth in q);
  work := string(wk in q);
  if author[1..10] = '<auth>Shak'
  then true
  elif author = '<auth>Milton</auth>' and
       ['P.L.', 'P.R.', 'Samson', ...] has work
  then true
  elif author = 'Spenser' and work = '<wk>F.Q.</wk>'
  then true
  elif (work = 'Bible' or author = 'Bible') and
       (abs(year(string(qdate in q))) - 1611) < 3
  then true
  else false;
  fi;
end;
```

Figure 5. A procedure which selects *OED* entries for the *NSOED*

3. Some Lessons

3.1. UNIX is a Good Place to Start

When we first received the *OED* data, we were apprehensive about the results of asking the UNIX file system to deal with 400-plus Mb. files. So far, the news is all good. No kernel or user component has yet complained about these files purely on the basis of size. Also, sequential processing of these files is sufficiently fast that most of our tools are CPU-bound. However, there are a couple of unanswered questions:

1. How well would a System V file system hold up under this sort of strain? To date, we have only worked with 4bsd and Sun UNIX variants.
2. Should we be using the file system at all? Needless to say, we don't use stdio. Our I/O exhibits some locality on some occasions, but probably not the kind that the file system code expects. One of the things we need to try is putting the data on raw disk partitions.

3.2. Descriptive Markup and SGML

Descriptive markup [Coombs] is an idea whose time has come. Future uses of electronic text are predictably unpredictable. The only scenario in which procedural markup such as troff or T_EX is valid is when creating a document which the author is certain will only be used for presentation on paper. Are there any such documents outside the domain of literature that are worth writing?

Unfortunately, the effort to standardize descriptive markup, namely SGML [ISO], has taken some dubious turns. While the idea is good, the standard is very complex and difficult to use. Some of the features and characteristics specified in it seem of questionable usefulness or even potentially harmful.

Fortunately, descriptive markup is a good enough idea that presumably it will survive this attempt to kill it with standards.

3.3. Prescriptive and Descriptive Grammars

One of the problems with SGML is that it attempts to impose a prescriptive grammar on the document creator. Chomsky demonstrated the problems with imposing prescriptive grammars on natural language in the 1960's; the New *OED* project has repeatedly failed to impose them on the dictionary and on other human-created documents. The language, and in fact the cognitive process, is of such complexity that an attempt to impose a prescriptive grammar on the authorial process may amount to intellectual impoverishment.

Does this mean that we are unable to impose any structure on text? Such a contention is absurd, and our software in fact makes use of the implicit structure inserted by the author. We extract this structure and build a hierarchical index to it, allowing us to query a structure as if there were some underlying grammar, without requiring that the grammar be fully specified. This structure looks and feels much like a grammar; is it proper to call it a descriptive grammar?

3.4. Windowing is Too Hard

The user I/O requirements of our software seem modest to us. We want to display text in rectangular windows in a variety of fonts (perhaps with some pre-cooked graphics). We want to respond to window resize events, to process a few point-and-click events intelligently, and on occasion to show some menus. We do *not* want to concern ourselves with look-and-feel, since any text database application is going to have to co-exist with a variety of other applications. We have no need for animation, exotic input modes, sophisticated colour, or anything else particularly fancy.

Despite this, we spend a totally excessive amount of time wrestling with X.10, X.11, font converters, incompatible binaries, NeWS LiteWindows, the SunView notifier, and many other things that are simply not of interest. Furthermore, every time we use one more window system call, we run a serious risk of ensuring that our software will never run on a Macintosh, or on a NeXT, or under OS/2. This is much harder than it ought to be. Perhaps in a few years the toolkit and look-and-feel dust will have settled, and there will be a straightforward way to do this with reliably portable results.

3.5. Typography is Important

Publishers have had several centuries of experience using typography to make complex text comprehensible. We on the project have yet to find a better way to solve this problem, but are somewhat crippled by the limitations of current font technology. There is hope that the PostScript model will improve this situation.

3.6. '\n' Considered Harmful

One of the reasons for the success of UNIX is the absence of "records" and the freedom and flexibility given by the byte-stream model for data access. In particular, UNIX and its utilities have been the system of choice for dealing with text data.

Unfortunately, most of those utilities have proven useless with our structured text data, simply because we do not choose to include newline characters. The semantics of newlines in free-form UNIX text files are very fuzzy (witness the troff \c operator) and our data is far from free form. In a highly structured text full of deeply nested tags, there is simply no correct, meaningful place to put newlines. Furthermore, if you want to put them every 70 characters or so just so you can use *sed* on the OED, you add 5.7 million characters to the text, each one a red herring. Alternatively, newlines could replace blanks, thereby complicating all programs that process the text. But this solution cannot guarantee a bound on line lengths without restricting the local density of blanks.

Instead, we run the text through a filter to insert newlines, use the UNIX tool, and then reverse the filter. But this kind of waste — writing programs to deal with incompatible "record" formats — is exactly the kind of thing that drove us away from IBM and VMS! (A tip of the hat here to *lex*, which processes arbitrary streams of text correctly).

Having newlines in text confers two benefits: you can use the *vi* editor, and you can use the "~" and "$" constructs in regular expressions. Perhaps neither of these is worth the loss of generality and the irritation of having to deal with records.

The obvious question is, then: "Where to break the lines?". The answer is "It depends." The editors used to update the data and the typesetting systems used to display it will distribute the text among lines in a manner appropriate to the task at hand; attempting to anticipate this at file creation time is futile.

4. Conclusions

Text processing has always been central to UNIX culture. For many years, the UNIX environment has provided the right paradigm (filters operating on byte streams) and the best tool set (*awk*, *grep*, *sed*, *troff*, *pic*, *tbl*, ...) for dealing with text data.

We now know what the next important step forward in this area will be: the intelligent manipulation, searching, and processing of *structured* text. Descriptive markup is an important first step in this direction. The crucial next step is the development of the kind of clean theoretical framework and well-stocked toolkit that has been available for traditional text processing.

It is to be hoped that UNIX will remain the system of choice for managing text data. For this to happen, the community needs to start working now on the theory and tools appropriate for structured text.

5. Acknowledgements

This paper owes an immense debt to Professor Frank Tompa, who edited it comprehensively, removed several *faux pas* and suggested valuable additions. Also of value were the comments of Donna Lee Berg and the USENIX referees.

Of course, the most important acknowledgement goes to the men and women who built the *OED* and to my co-workers at Waterloo who did the creative work described here.

References

[Coombs] Coombs, James H, Allen H. Renear, and Steven J. DeRose *Markup Systems and the Future of Scholarly Text Processing. Comm. ACM*, 30, 11 (1987), 933.

[Gonnet87] Gonnet, G.H. and Frank Wm. Tompa *Mind your grammar: a new approach to modelling text. Proc. 13th VLDB* (1987) 339-346.

[Gonnet88] Gonnet, G.H. *Efficient Searching of Text and Pictures.* Tech. Rep. OED-88-02. UW Centre for the New OED, Feb. 1988.

[ISO] ISO (International Organization for Standardization) *8879 Information processing — text and office systems — Standard Generalized Markup Language (SGML).* (1985).

[Kazman] Kazman, R.N. *Structuring the text of the Oxford English Dictionary through finite state transductions.* Tech. Rep. CS-86-20, Computer Science, University of Waterloo, (June 1986).

[Knuth] Knuth, Donald E. *The Art of Computer Programming.* Addison-Wesley, 1973, III, 490.

[Murray] Murray, J.A.H., H. Bradley, W.A. Craigie and C.T. Onions *The Oxford English Dictionary.* Oxford at the Clarendon press, Oxford, England, 1928.

[Raymond] Raymond, Darrell R., and Frank Wm. Tompa *Hypertext and The New Oxford English Dictionary. Comm. ACM*, 31, 7 (1988), 871.

Implementation of Dial-up IP for UNIX Systems[1]

Leo Lanzillo
CSNET Coordination and Information Center
BBN Systems and Technologies Corporation
Cambridge, MA 02238
617-873-2777
leo@sh.cs.net

Craig Partridge
BBN Systems and Technologies Corporation
Cambridge, MA 02238
617-873-2777
craig@nnsc.nsf.net

Abstract

CSNET has developed a software package to support the sending of Internet Protocol (IP) datagrams over dial-up phone lines. This driver can automatically establish and disconnect phone calls as IP traffic dictates. This code works in binary-only BSD systems.

1. Introduction and Motivation

As the proliferation of computers, workstations, and local area networks continues, the demand for an inexpensive, simple, and reliable method of connecting computers and networks also grows. One of the least expensive methods of connecting computers is to use dial-up phone lines. This approach results in a true pay-for-usage fee service, but until now the use of dial-up phone lines has been restricted to application-specific protocols, instead of more general networking protocols that could be used by multiple applications.

Dial-up IP is a solution to this problem. Because of its dial-up nature, its cost is directly related to usage. It requires only modems and serial interfaces, and, most important, it provides Internet Protocol (IP) connectivity between the two endpoints. IP connectivity allows the host administrators to link computers together using any Internet protocol, including TCP and UDP, and at higher levels, FTP, TELNET, *rcp*, *rlogin*, and NFS.[2] In addition, the system's use of dial-up links allows for the on-demand joining of different networks by connecting IP gateways to form an internetwork of computers.

1.1. A New CSNET Service

We view dial-up IP as an enhanced replacement for CSNET's PhoneNet service. CSNET is a logical network, a metanetwork, which, in addition to information and consulting services, offers two distinct types of communication services. The "top-of-the-line" service is full-time IP connectivity with the Internet through the DARPA-approved gateway, *relay.cs.net*. Connectivity can be provided via

[1] UNIX is a registered trademark of AT&T. CSNET is operated by BBN Systems and Technologies Corporation under a subcontract with the University Corporation for Atmospheric Research, which operates under contract with the U.S. National Science Foundation. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of BBN, UCAR, or the NSF.

[2] Don't try this at home. Currently, commercial NFS systems do not run well through gateways or over slow links, however an experimental version of NFS that has been enhanced with Jacobson's algorithms has been shown to work quite well over complex internets.

X.25 over a public data network [1], via Cypress [2], or via a leased data line to the CSNET CIC. The monthly costs of such arrangements are sufficiently high that they are usually of interest only to high-volume users (i.e., users requiring more than two to four hours of connectivity per day).

The other CSNET communication service is PhoneNet, which provides electronic mail connectivity to over one hundred universities and research centers. PhoneNet is a star-shaped, store-and-forward mail network that uses dial-up phone lines and modems to transfer mail between remote sites and *relay.cs.net*. *Relay* acts as a gateway to the Internet and to other PhoneNet sites. These sites periodically call or are called to deliver mail. The average delivery time via PhoneNet is less than twelve hours.

PhoneNet is limited in that it is used only for mail. Many of the CSNET clients have asked for access to the wider range of Internet services available via TCP/IP, in particular, anonymous FTP. There have also been occasional complaints about the limited error recovery ability of the PhoneNet protocol.

Dial-up IP is an intermediate offering that combines the positive aspects of CSNET's other services: the robustness and reliability of full-time IP, the efficiency and cost management aspects of PhoneNet, and ready availability. Dial-up IP is also more secure than a full-time IP connection, both because there are a number of security checks that must be passed before a connection is established and because the line is only up for limited periods of time.

1.2. Design Goals

We had three major design goals. Dial-up IP had to be easy to install, transparent to users, cost-effective and moderately secure.

Dial-up IP had to be easy to install and to configure. Inexperienced system administrators needed to be able to install software quickly and to not have to worry about it. For these reasons we designed dial-up IP to fit cleanly into the UNIX kernel, requiring modifications only to configuration files and a couple of include files.

Given the diversity of the CSNET community, and the fact that many sites are universities with high-turnover student help, we had to design dial-up IP to be simple to use, or better yet, transparent to the users. Other dial-up systems require user intervention to establish a phone link.[3] We wanted to develop a version of dial-up IP which makes the connection establishment transparent to the user. This approach has a further advantage of saving the user from having to know which network a particular host is on. Dial-up hosts, or hosts on LANs accessible only through dial-up gateways, appear as accessible as hosts on permanently-connected networks.

Furthermore, we wanted dial-up IP to be cost-effective. We wanted it to use the line resources efficiently by filling the line bandwidth when a phone line was active and minimizing phone charges by terminating phone lines quickly when they were no longer in use.

Finally, we wanted dial-up IP to be moderately secure. In particular, we wanted sites to be able to control when dial-up links were established and who was able to establish them.

We believe we have achieved these goals.

2. A Datagram's Journey Through Dial-up IP

To illustrate how dial-up IP works, a brief and slightly simplified description of an example dial-up IP interaction follows.

Imagine that *big-college.edu* has a dial-up IP link to *relay.cs.net*, and that a user on *big-college.edu* wishes to FTP a file from some other Internet site.

---

[3] Just as this paper was going to press the authors learned of the existence of another dial-up IP implementation, developed at the University of Tokyo. We do not have enough information about the Univ. of Tokyo implementation to compare it with ours. It may also support on-demand connections.

When *ftp* is invoked by the user, it calls on TCP to establish a connection to the other site. TCP, in turn, calls upon IP to send a TCP SYN datagram (a synchronizing segment used to open connections) to the remote site.

When IP receives a datagram it consults its routing tables to find out which network interface to send the datagram out on. In this example, the routing tables point to a dial-up IP interface, *du0*. The IP layer then queues the datagram on the outbound queue of *du0*.

When a datagram arrives on *du0*'s queue, the dial-up interface code checks to see if a phone call is currently in progress to *relay.cs.net*. If a call is in progress, the datagram is sent over the line. However, if a call is not currently in progress, the dial-up interface must find a way to dial the phone.

Unfortunately, the interface layer of the kernel is not the best place to insert code to dial a modem. Dialing a modem takes several seconds and the interface code in the kernel is running at a high interrupt level. To get around this problem, we created a pseudo-device, */dev/dial-up*, for requests to dial the phone. */dev/dial-up* maintains a queue of requests for modems to be dialed. *Du0* requests a phone call by writing a request structure into the queue. A daemon, *diald*, reads the requests from the queue, finds an available modem, dials the remote host, and then logs into a special account. (Note that we use standard UNIX *login*; the account is special because it has a login shell that brings up dial-up IP). Once the login is completed, the SLIP protocol [7] is activated on the line. At this point, the local dial-up interface is notified that it can start passing datagrams over the connection.

On the remote end, when the datagram is received from the dial-up line, it is passed up to the IP layer, like a datagram from any other interface.

After the phone call is established, the dial-up IP driver watches the line. If the line is idle for too long the driver initiates a hangup. The maximum idle time is run-time configurable and is usually set to be a few minutes.

3. Details of the Kernel Implementation

3.1. The Dial-up Interface

Dial-up IP looks like a standard BSD interface to the higher layers. The interface code is a substantially enhanced version of the SLIP driver written by Rick Adams and distributed with BSD 4.3. In addition to handling the SLIP protocol, the driver now includes code to check on the status of the line (and request a phone call if the line is down), code to time out the line if it is idle, and extensive support for monitoring tools.

One complication in the interface is that a phone call may fail, leaving outbound datagrams sitting in the interface queue. We handle this problem by having the driver timeout the connection attempt if it takes too long and discard all queued datagrams. (Thankfully, IP permits datagrams to be discarded if they are undeliverable).

Another source of trouble comes from concurrent connection attempts (phone calls from both ends of the connection at the same time). This is handled by having the driver sleeping on a flag, and allowing it to be awakened either by a successful outgoing call, or a successful incoming call. The other call is then locked out and will fail.

3.2. The Dial-up Pseudo-Device

The driver for the pseudo-device, */dev/dial-up*, is new and comprises just a few lines of code. The one difficulty in writing the pseudo-device code was to avoid race conditions between the high-interrupt level interface driver and the application-level daemon. The record passed to the daemon contains address information and as well as part of the IP header. This gives *diald* enough information to make selective decisions about when to dial the phone (see below).

3.3. Routing

Routing for dial-up IP is simple. The dial-up IP interface looks like a standard point-to-point link to the rest of the kernel. Thus any datagram that can be sent over a point-to-point link can be sent over dial-up IP, including datagrams to networks that are on the other side of a dial-up link. Multiple dial-up

IP links on a single host are feasible, although we recommend using dial-up IP only in a star topology; establishing multi-hop connections takes too long.

One small routing complication we encountered was that the BSD kernel checks to see if an interface is in the UP state before queuing datagrams on it. A dial-up link may not be up before it receives a datagram, so we had to fool the kernel into thinking that the interface was up, when in fact, no call was active. We solved this by simplying leaving the interface state hardwired as UP, and keeping track of whether we need to make a phone call with another flag maintained by the interface code.

3.4. Support for Multiple Platforms

The dial-up IP software package provides full IP support over dial up phone lines. It runs on Ultrix, Sun and BSD UNIX systems and requires a modem and a serial port on the computer. This software communicates with other computers that support dial-up IP or at least some version of SLIP. For example, dial-up IP works with Phil Karn's KA9Q SLIP package for IBM compatible PCs. We are currently investigating using it with Sun's dial-up PC-NFS package.

4. Support Applications

Dial-up IP requires some programs that run as user processes. These programs take care of call set up, interface configuration, monitoring and so on. These programs also make use of configuration files which specify phone numbers, devices, account names and connection-time restrictions. User level programs have also been developed for monitoring the dial-up IP link at both the IP and serial line level. Finally, utilities have been written for examining and changing timeouts and other interface parameters.

4.1. The Dialing Daemon

The key support program for providing transparency to the user is the dialing daemon, *diald*, which dials the modem when dial-up interfaces need a connection established.

Diald is partially derived from the existing PhoneNet code that knows how to dial telephones. The dialing code reads a script that lists the commands necessary to establish a connection and log in. The command language for scripts is sufficiently diverse that *diald* can establish a link not only over phone lines, but also through *telnet*, X.25 PAD and various switching devices such as port-selectors and even PC-Pursuit.

Diald also supports facilities that restrict access to the dial-up sites. Recall that the */dev/dialup* device passes up some information from the IP header of the datagram that caused the request to bring up the link. By checking this information with instructions in a configuration file, *diald* can refuse to initiate calls based based on the source or destination IP address of the packet, the time of day, or higher level protocols (ICMP, UDP etc). These facilities allow the host to filter requests for bringing up the line and give administrators greater control over costs and access to their network.

When a dial-up interface must make a phone call, it places a request in the */dev/dial-up* pseudo-device. When *diald* reads this request, it matches the IP destination of the request with a specific remote host. The daemon then performs the security checks on the IP protocol number, the source and destination addresses, and the time of day. If the request is legitimate, *diald* forks a child that runs the script to dial the phone number and perform the login sequence. At the remote (passive) end, a special shell is invoked that does further handshaking between the shell and the daemon child. Once the handshake is complete, both the child and the remote special shell do *ioctl()*s to activate their dial-up interfaces. (The interfaces are brought up by switching the *tty*s to a special line discipline.) The processes then sleep until the line is dropped. When the drivers time out the line, they send a SIGHUP to the controlling processes, at which time the processes clean up and exit.

Because the daemon-child and the special shell sleep instead of dying, the UNIX kernel believes that the *tty* port is still in use. This prevents other *getty*s from running on the passive end of the connection and prevents the line discipline from being reset on both ends.

4.2. Manually Starting or Ending a Connection

In order to provide greater control to system administrators, we provide a utility called *upline* for manually establishing a connection to a particular dial-up site. The *upline* program can be run by hand or from shell scripts, and it takes a hostname or IP number as an argument. It operates much like the *diald* daemon to establish a link.

In order to save phone charges, the dial-up IP driver will timeout inactive lines and terminate connections, which saves the user or administrator from dealing with disconnects. But if they wish, there is also a program, *downline*, which allows administrators to take down a link manually.

4.3. Access Control

Currently, the only limits on the access and use of dial-up IP links are the checks done by *diald* before establishing a connection. We are considering enhancing the dial-up IP access controls to filter out unwanted traffic while the link is up. The justification for such filtering is both that sites may wish to protect themselves from unauthorized IP access from outside and, more importantly, that the dial-up links only be used by traffic that the site wants to pay for.

So far, the best algorithm appears to be to maintain a hash table of remote addresses. Under this scheme, a remote IP address is accredited by receiving a datagram from the dial-up IP site destined for the remote address. Thus, whenever a datagram is received from the link, the outbound IP address is added to the hash table, and any time a datagram is presented to be sent out the link, the driver would first check to confirm that the source IP address was in the table. Given that most links are only up for a brief period (usually much less than one hour) and the number of remote sites with which it exchanges data in one session is small, the hash table of accredited addresses should not need to be very large and thus it would be possible to put this hash table in the kernel.

5. Services and Facilities

Dial-up IP provides a basic connection to *relay.cs.net* but applications programs must be run to accomplish any real work. One problem is get these programs running (on remote systems) after a phone line is brought up. Since one of the main functions of dial-up IP in the CSNET community is to handle mail delivery, we have developed software that allows dial-up sites to request mail and other services from *relay.cs.net*. (All Internet mail to the site can be directed to *relay.cs.net* for queuing through judicious use of the domain name system's MX resource records [5].)

This software is based on a client-server model in which the client is able to ask the server to start mail delivery via the Simple Mail Transfer Protocol (SMTP) [6] to a particular site, or to get usage data for a site, or to provide access to the BIND nameserver. We believe a number of services remain which would be useful to dial-up IP users, and this client-server program provides a platform for these additional services.

Because dial-up IP is a pay-for-usage service, we provide tools for monitoring the usage at both the calling and the called end. Each call is logged, and the duration of the call is recorded so that administrators can monitor usage of the service. In addition, there are facilities for dynamic monitoring of the link at both the IP level and and at the serial line level. A monitoring package is available that allows lines to be monitored locally or remotely, and can display statistics (using curses) on the amount of traffic, utilization, number of characters, number of errors, and so on. There is also a tool for examining and dynamically modifying some of the dial-up parameters such as the timeout values and flags.

6. Performance

When we started work on dial-up IP, we were concerned that its performance might not be satisfactory. In particular, we were concerned about the so-called "slow-line problem." The slow-line problem affects communications over transmission lines whose speed is sufficiently low that the size of a datagram noticeably influences its transmission time. For example, IP datagrams typically vary in size from about 40 to 2048 bytes. On a 10Mbit (1.25Mbyte) Ethernet, any datagram in this size range can be transmitted in a fraction of a second. But on a 2400 bps telephone line, while a 40 byte datagram takes a sixth of a second, a 2048 byte datagram takes nearly 7 seconds.

The dramatic variation in transmission times can confuse TCP implementations. TCP uses an estimation of the network transmission times to decide when to retransmit a piece of data. On a slow link it is possible for a TCP implementation that has a low transmission estimate (based on transmission times of small datagrams) to send multiple, unnecessary retransmissions of large datagrams which clog up the link. In early 1988, most TCP implementations used time estimation algorithms that did not deal well with large variations, and informal reports suggested that in some scenarios, the slow-line problem caused 90% of the nominal bandwidth to be lost. As a result, we feared that the slow link problem would make dial-up IP uneconomical.

Thanks to the work of several researchers, primarily Van Jacobson and Phil Karn [3,4], TCP time estimation algorithms have improved dramatically in the past year. By the time we tested our TCP implementation (which incorporated these new time estimation algorithms) over the dial-up IP links, we achieved throughput rates of over two-thirds of the line speed.

One unexpected problem with using dial-up IP is that it can take over one minute to make a phone call and establish a dial-up IP link. This presents a problem for many current implementations of TCP that use a timeout of only 30 seconds for the initial connection setup (the time they are willing to wait for an acknowledgement to the first SYN datagram sent out). We found that we had to increase the TCP connection timeout value to be 75 seconds for dial-up IP to work smoothly. To ensure that all TCP implementations use equally high timeout values we have asked the Internet Engineering Task Force, which is in the process of developing a set of standards for hosts running TCP/IP, to require a longer TCP connection timeout.

7. Conclusion

We believe that dial-up IP is an excellent replacement for CSNET's PhoneNet service. In addition to cost savings, we have experienced fewer problems with PhoneNet sites using dial-up IP. Less overhead is required to support each site. There are fewer problems establishing the connection. TCP has proved better able to deal with transmission problems than PhoneNet. The biggest gain is that SMTP is more robust than the PhoneNet protocol so that there are fewer stuck messages which clog the mail queues.

More generally, we believe this service is well suited for linking a number of remote stations or isolated LANs to each other, particularly for sites which require only infrequent IP access to computers at other sites. One limitation of dial-up IP is its long link setup time. Because link setup currently takes a long time, TCP connections cannot currently establish a connection over a path that has multiple dial-up links. (Note that once a dial-up link has been established, it behaves just like a dedicated line).

References

[1] D.E. Comer and J.T. Korb, "CSNET Protocol Software: The IP-to-X.25 Interface," *Proc. ACM SIGCOMM '83*, pp. 154-169, Austin, Texas, March 1983.

[2] D.E. Comer and T. Narten, "UNIX Systems as Cypress Implets," *Proc. 1988 Winter USENIX Conf.*, pp. 55-62, Dallas, Texas, February 1988.

[3] V. Jacobson, "Congestion Avoidance and Control," *Proc. ACM SIGCOMM '88*, pp. 314-329, Stanford, California, August 1988.

[4] P. Karn and C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols," *Proc. ACM SIGCOMM '87*, pp. 1-7, Stowe, Vermont, August 1987.

[5] C. Partridge, *Mail Routing and the Domain System; RFC 974*, Internet Working Group, Requests for Comments, No. 974, DDN Network Information Center (NIC) at SRI International, Menlo Park, California, January 1986.

[6] J. Postel, *Simple Mail Transfer Protocol; RFC 821*, Internet Working Group, Requests for Comments, No. 821, DDN Network Information Center (NIC) at SRI International, Menlo Park, California, August 1982.

[7] J. Romkey, *A Nonstandard for Transmission of IP Datagrams Over Serial Lines: Slip; RFC 1055*, Internet Working Group, Requests for Comments, No. 1055, DDN Network Information Center (NIC) at SRI International, Menlo Park, California, June 1988.

A UNIX Implementation of the Simple Network Management Protocol

Wengyik Yeong
yeongw@nisc.nyser.net

Martin Lee Schoffstall
schoff@nisc.nyser.net

Mark S. Fedor
fedor@nisc.nyser.net

NYSERNet Incorporated
Rensselaer Technology Park
165 Jordan Road
Troy, New York 12180

Abstract

The many choices in network products available today from a diversity of vendors clearly indicates the need for a single, non-proprietary solution to the problem of managing all the entities in the existing TCP/IP Internet. The Simple Network Management Protocol (SNMP) is a solution to the network management needs of Local Area Networks, as well as the greater Internet.

This paper describes the design and implementation of software conforming to the SNMP for computer systems running Berkeley 4.2/4.3 and related operating systems. The implementation described herein includes all software necessary for the management of machines running such operating systems: a network management station for UNIX, and an SNMP agent (server) intended to run on similar systems.

1. Introduction

The management of network entities involves the monitoring of network state in real time, the collection of data on various characteristics of networks, and the resolution of network problems before they reach catastrophic proportions. The increasing sophistication and heterogeneity of network elements, coupled with the explosive growth of the TCP/IP Internet in recent years has overwhelmed the abilities of network managers to accomplish these tasks. The Simple Network Management Protocol [1] was designed to be a solution to network management needs.

2. Origins

At the end of 1986, with the emergence of the NSFNET backbone and mid-level networks, it became apparent that network management needs and issues would have to be addressed. In March of 1987, individuals representing a user base from Cornell, RPI, the University of Tennessee, NYSERNet, and SURANet joined with Proteon Inc.

and began development on the Simple Gateway Monitoring Protocol (SGMP) [2,10].

Collectively bringing significant experience in TCP/IP networking, this group was able to specify the SGMP by mid-1987. Several working prototypes were completed and freely distributed, and Proteon Inc. was shipping SGMP in production by the fall of 1987. January 1988 saw the first production network management station products from NYSERNet Inc. and the University of Tennesse at Knoxville. SGMP then saw wide use by the mid-level networks such as NYSERNet, SURANet and BARRNet, the new NSFNET backbone supplied by IBM, and on numerous campus LAN's. SGMP growth was astronomical and time saw the protocol used in areas such as the monitoring of terminal servers. This was an area of application that had not been conceived by the protocol authors.

In March of 1988, the Internet Activities Board (IAB) called a meeting chaired by Dr. Vinton Cerf to coordinate network management efforts. The outcome and recommendations of this meeting are documented in [3]. SGMP was chosen as the "short term" standard and would share a common information base with a future OSI effort: the Internet Structure for Management Information (SMI) [5] and the Internet Management Information Base (MIB) [4]. This common information base would provide for the future migration of network management to OSI. This led to an evolution of the SGMP: the Simple Network Management Protocol (SNMP) [1]. SNMP would address network entities beyond gateway/routers and would do more than monitor; it would control.

3. Protocol Overview

The SNMP is a transaction oriented protocol that is stateless, allowing for the direct querying of network elements. The timeliness of data collected by such means makes the SNMP ideal for the real-time monitoring application and firefighting; the low network overhead incurred in making such queries also provides an inexpensive way of gathering data on network characteristics.

The information base managed by the SNMP is that defined by the work of the IETF MIB working group [4,5]. It is shared with the OSI network management effort, and in its present form attempts to describe a model for the management of entities belonging to layers 1-3 of the OSI reference model. A preliminary effort was also made in this information base to model the management of layer 4.

The SNMP is based on the SGMP, a protocol that had been implemented and deployed in production for a year at the time of the SNMP's conception: the SNMP was based on a protocol that had seen wide use in the internet and incorporated real-world experience in network management.

4. Technical Objectives

SNMP was engineered to fulfill a set of goals deemed important by the authors and a significant portion of the Internet community. Before work on the protocol began, attention focused on the needs of the user community. Low latency, survivability, and idempotence drove the authors to a connectionless transport protocol, UDP [6].

A decision was made not to base the SNMP on connection oriented transport protocols such as TCP [7] as it was felt that for the purposes of network management, such protocols incurred an unnecessary amount of overhead. Connection oriented protocols have a high latency due to the work involved in establishing and tearing down connections. Common TCP implementations may require as many as five packets to transfer a single octet of data (during connection setup and teardown). As most management actions and responses will fit into one UDP packet, entire SNMP operations need only two packets to complete. Latency and network load is hence reduced. In networks of high congestion, excessive errors, or large instability, management

needs increase. It is at such times however that it would be desirable for the application to have control over such communication parameters as timeouts and retrys so as to allow maximal adaptability to network conditions. Modern, connection-oriented transport protocols (including TCP) do not provide such great degree of control to the application, but UDP will, ensuring the survival of the critical network management function.

Another objective of the SNMP was to place protocol complexity in the network management station (NMS) and not the remote agent/server. This would create a protocol that would incur minimal overhead on managed network elements.

A conscious choice was made to use a widely-accepted, standard, method to encapsulate SNMP Protocol Data Units (PDUs). This lead to the adoption of Abstract Syntax Notation One (ASN.1) [8,9] for such purposes. Such a decision would also ease the transition to OSI style network management.

5. Implementation Overview

This implementation of the SNMP was completely written in the 'C' programming language, and runs on machines with Berkeley 4.3 or a related operating system.

Modularity of the implementation was an objective. The implementation is divided into three major modules: a protocol library, an SNMP agent and SNMP network management station software. Each of these major modules are further subdivided into smaller modules which are described in the respective sections of this paper.

This implementation can also be divided into two 'layers'. The lower layer consists of the SNMP library, and is generally responsible for details pertaining to authentication, communications and protocol processing. The upper layer consists of SNMP network management station applications and the SNMP agent.

As a result of the effort put into ensuring the modularity of the implementation, the SNMP network management station applications, and to a lesser extent, the SNMP agent, is insulated from a great deal of protocol and communications processing.

It was also an objective of this implementation that it be portable. To ensure that the implementation would run on a diverse set of hardware architectures, use of the 'C' language 'int' type was restricted. Wherever an ambiguity could have resulted due to differences in machine architecture and/or word size, the 'C' language's 'long' and 'short' types were used. In addition, great care was taken to ensure proper byte order, with all applications using host byte order, and the library converting to, and using network byte order wherever necessary.

This implementation utilizes the Berkeley IPC 'sockets' primitives for the purposes of communications. Hence all communications in the implementation are conducted synchronously. However, it is expected that few changes will have to be made, all in the SNMP library to allow the implementation to use other communication methods.

6. The SNMP Library

The objective of the SNMP library is provide a set of building blocks to allow for the modular development of both an SNMP agent and a network management station. It is made up of: an encoder and decoder for Abstract Syntax Notation One (ASN.1) [8,9], communications functions, authentication functions, and a variable converter to convert between symbolic variable names and the object identifiers used to identify objects in the Internet Management Information Base [4].

The ASN.1 encoder and decoder in the library convert between the SNMP's Protocol Data Units (PDUs), which are encoded with the Basic Encoding Rules [9] of ASN.1, and the 'C' language structures used internally by the agent and the network management station. Both the encoder and decoder are structured as recursive descent

parsers with the source language of each serving as the target language of the other.

The decoder operates on serialized BER structures, viewing its input as a stream of octets. It first parses out the version number and community name of the packet received, then determines the type of PDU within the SNMP packet received. The decoder will then cast the buffer provided by the user (of the decoder) to the appropriate 'C' language structure and fill it in appropriately as it parses the remainder of the ASN.1 packet received.

The encoder performs the function opposite to the decoder, converting from a 'C' language structure to a packet containing ASN.1 structures. It accepts a buffer containing the 'C' language structure to be encoded and the type of message contained in the buffer as input parameters. The encoder then serializes the contents of the buffer into the ASN.1 language according to the Basic Encoding Rules of ASN.1 [9] producing a PDU suitable for transmission.

The purpose of the SNMP variable converter is to map between user-friendly character strings used to represent the objects in the Internet MIB [4] and the object identifiers used within the SNMP's PDUs. The variable converter consists of three functions, to build the initial mapping, to convert between symbolic character strings and object identifiers, and to do the reverse mapping.

The initial function reads the '/etc/snmp.variables' file, building a mapping from the contents of the file. As the variables in the MIB are inherently hierarchical, an n-ary tree is used to represent the mapping internally, with each node of the tree mapping between an object subidentifier and its symbolic representation. For example, the MIB object 1.3.6.1.2.1.1.1 which can be represented symbolically as _iso_org_dod_internet_mgmt_mib_system_sysDescr would have a tree representation consisting of a node that maps between the string "iso" and the integer 1, a node that maps between the string "org" and the integer 3 and so on, ending with a node mapping between the string "sysDescr" and the integer 1.

The two other functions that make up the variable converter then traverse the tree to convert between symbolic representations and object identifiers. The conversion from symbolic representation to an object identifier is straightforward: the tree is walked with the conversion function comparing nodes at each level of the tree to the corresponding component of the variable to be converted. The reverse conversion, from object identifier to symbolic variable involves the somewhat unusual use of the tree structure however: the subidentifiers in the object identifier are used to index into the pointers to each nodes children, thus allowing the location of the correct child (and thus the correct symbolic equivalent) in constant time.

As the only authentication currently available in the SNMP is by means of the community name, the functions to perform authentication within the SNMP library are null functions. These functions do not perform any mapping, encoding or decoding for authentication purposes when invoked. In the future, when a non-trivial authentication scheme comes into use, the authentication module in the library will be enhanced to perform non-trivial authentication.

The communications functions in the SNMP library tend to the details involved in using the Berkeley IPC 'sockets' primitives. The communications functions invoke the encoder/decoder and the authentication functions as appropriate upon receipt/transmission of SNMP PDUs when called by the library user.

A pair of functions that basically function as 'top-level' functions to invoke ASN.1 encoding/decoding and authentication only is also included for the use of applications that wish to take care of details of communication for themselves.

The high-level interface to the library is provided by a common application interface to the communications functions in the SNMP library. This interface is intended to be used by applications which do not wish to be concerned by details of the

communication and details of the SNMP, and provides a means of writing applications that independent of the management protocol beneath to a great extent. Further details on the implementation can be found in [11].

7. The SNMP Agent

The SNMP agent/server is the portion of the implementation which resides on the network entity being managed. The agent answers requests for information, performs set operations, and sends out the appropriate event traps. The agent can be embedded in the network entity, such as a dedicated router, or it can be a user program in the UNIX style of daemons. This implementation of an SNMP agent was developed to run as a UNIX network daemon.

The agent does not use the applications interface for SNMP communications. Instead, it calls the needed parts if the SNMP library directly. The agent controls socket sends and receives internally, outside of the SNMP library.

7.1. Design Goals

Certain criteria had to be met in the designing of the SNMP agent for UNIX. First, the SNMP specific code had to be independent from the the object look-up code. This was required in order to make portability much easier as kernel look up methods are not the same in all UNIX derivatives. Second, the agent had to be relatively small and efficient. Thirdly, as many objects as possible in the Management Information Base (MIB) [4] had to be supported *without* making any UNIX kernel modifications. This is so the agent could be used in an environment where the network manager only had a binary only UNIX operating system.

7.2. Internal Structure of Management Information Base (MIB) Variables

The objects specified in the MIB document [4], which are supported by the agent, are stored in a tree structure. The objects are always found at the leaf nodes. The leaf node for each object includes a pointer to the appropriate look-up function, a pointer to the appropriate set function, and a flags word. By having the function pointers in the leaf nodes for each object, you achieve the design goal of having independence between the protocol and the action functions as one routine performing protocol specific actions can be used on all objects supported in the tree. Also, object look-ups cost the order of the depth of the tree, thus providing an efficient look-up method.

7.3. Variable Searches

The look-up of an object is done by using each component of the object identifier as an index down into the tree. For example, the iso_org_dod_internet_mgmt_mib_system_sysDescr object has an identifier of 1.3.6.1.2.1.1.1. Each part of this object is used as an index into the tree. After the final component is used, one should now be at the proper leaf node. The object value can now be retrieved and sent back to the requestor.

If, after using the object identifier to index into the tree, a leaf node is not present and the requestor has issued a SNMP get-request, an error is returned as stated in the SNMP protocol specification. If, after using the object identifier to index into the tree, a leaf node is not present and the requestor has issued a SNMP get-next request, a depth first search is started from the current index into the tree in order to find the next valid object in the variable tree. If no valid objects are found, an error is returned as stated in the SNMP protocol specification.

7.4. Extraction of Values

Values for the objects supported in the MIB are retrieved in a number of ways. The most widely used method is extracting the values from the UNIX kernel. This is done through the */dev/kmem* interface. Certain symbols in the namelist of the kernel are used to access kernel data structures. These symbols include: ifnet, rtnet, rthost, ipforwarding, ipstat, udpstat, tcpstat, icmpstat, arptab, boottime, and tcb. Another method in which the agent retrieves values for its supported objects is via a configuration file. On start up, the agent reads the configuration file and stores the information in its internal information lists. The look-up routines for certain objects access these information lists to get values.

7.5. SNMP Set Operations

The agent performs set operations as stated in the SNMP specifications. The SNMP specifications state that each object in a set request packet must be set as if set simultaneously. The atomicity of the set operation proved to be an interesting implementation exercise. Should the setting of any one object fail in a set packet which had multiple objects in it, the previous values of the objects just altered had to be restored. If all the objects in the SNMP set request packet could not be set properly, the network entity had to be left in the same state as it was in before the SNMP set request.

When a set request is received by the agent, the objects in the request are checked against failure conditions stated in the SNMP specifications. After each object is checked against error, it is added to a doubly-linked list of objects to be set. Once all the objects have been checked and added to this list, the actual set process begins. If, as the list is traversed, any set operation fails, the traversal stops and starts back up to the front of the list, replacing all previously altered objects with their old values. An error packet is then returned to the requestor as stated in the SNMP specification.

7.6. SNMP Traps

The agent generates SNMP trap messages as appropriate. A trap is a gratuitous message sent by the network management agent to the network management station when certain network events defined by the SNMP take place. The SNMP defines six traps: coldStart, warmStart, linkDown, linkUp, authenticationFailure, and egpNeighborLoss. The destination of the trap messages can be controlled by specifying a community of type "traps" in the agent configuration file. Trap messages can be sent to as many Network Management Stations as is specified in the agent configuration file.

7.7. Management of UNIX Daemons.

The UNIX kernel does not provide all of the information necessary to support all of the MIB objects. Some of the information is available in the UNIX daemons. The SNMP agent has the ability to communicate with UNIX daemons using interprocess communication. The agent speaks a simple protocol with the daemon in order to find out what objects the daemon supports. The advantage of this design over using the SNMP in every UNIX daemon which needs to be managed is that you do not burden the daemons with the task of encoding and decoding SNMP messages. Currently, only the UNIX routing daemon *gated* [12] has this capability. Plans call for this capability to be added to *named* [13,14] and *ftpd* [15]. The simple protocol used by the agent and daemons is still being developed and enhanced. Therefore, the protocol has not been published.

8. The Network Management Station

This implementation of the SNMP Network Management Station (NMS) was designed in the UNIX tradition of using the best-fitting tool for the task at hand: instead of being a single monolithic entity, the NMS was implemented as a collection of applications of varying sophistication and functionality.

The NMS was also designed to address all three facets of network management. Tools are included for both real-time network monitoring and firefighting, to enable network managers to respond to changes in network state in a timely fashion. In addition the NMS allows the collection of network statistics for the purposes of network planning, trend analysis and the detection of potential network problems.

In the interests of modularity, the applications that collectively make up the NMS are all written to the common application interface of the SNMP library (please see the section *The SNMP library*). This would allow the applications to run over other network management protocols as such protocols emerge and mature simply by making changes to the underlying library.

In accordance with the SNMP philosophy of keeping things simple, the applications that make up the NMS adopt a very simplistic method for error recovery: any error that occurs during the processing of a protocol transaction will result in the offending SNMP packet being discarded. Hence SNMP packets with incorrect request ids, community names, and non-zero error statuses will be discarded after the output of a possible error message. This simplistic approach to errors is justifiable as a discarded packet will only cause retransmission of the request, resulting in another reply from the SNMP agent.

There is no single paradigm of NMS function that serves to model the interaction between the SNMP NMS and SNMP agents resident on manageable network entities. Instead, it is recognized that functionality needed from the NMS under different situations may require different models of interaction.

The data collection tools in the SNMP NMS utilize a database oriented model whereby a single application runs in the background on the NMS hardware and logs data into a database (currently UNIX flat files, but plans call for integration with a database system in the future). As this overabundance of raw data often collected from a large number of network entities will almost always overwhelm a network manager, other NMS tools operate on the network management database to further process the data, providing summaries and reports that are more amenable to humans. The tools used for network analysis and planning are therefore of a 'background' nature, utilizing a database instead of directly interacting with SNMP agents.

In contrast to this database model, the SNMP NMS tools used for real-time monitoring of network entities interact directly with SNMP agents. It was felt by the implementors that interposing a database between the SNMP NMS tools to be used for this purpose and the SNMP agents residing on the network entities to be managed was not acceptable due to the need for NMS tools engaged in real-time monitoring to react quickly to changes in network state: having real-time monitoring occurring based on information from a database of past network behavior defeats the purpose of monitoring the network in real-time.

To aid in the rapid assimilation of data by network managers and operators, some of the tools that make up the real-time monitoring portion of the SNMP NMS are graphically oriented. In line with the desire for portability, a decision was made to avoid proprietary graphics packages and windowing systems. Instead, the graphical applications in the SNMP NMS was programmed to the 'C' language interface of the X11 Release 2 graphics package, a defacto standard in the UNIX community.

In comparison to the real-time monitoring tools in the NMS, the timeliness of information upon which firefighting tools operate is even more critical. Therefore,

much like the NMS tools used for real-time monitoring, the NMS tools used for firefighting interact directly with SNMP agents. Unlike the real-time monitoring tools however, NMS tools used for real-time firefighting operate under the direct control of a (human) network manager/operator due to the critical need for rapid reaction to network state changes during network fires.

9. Implementation Difficulties

While developing an implementation of the SNMP for the UNIX environment, it was not surprising to discover that UNIX lacks network management capabilities. For instance, the UNIX kernel lacks the instrumentation to support all of the manageable objects defined in the MIB [4]. Our implementation has shown that 78.37 percent (87 of 111) of the MIB objects can be collected from the UNIX environment in some way or another. 82.75 percent (72 of 87) of the MIB objects supported have their values taken directly from the UNIX kernel.

The UNIX operating system does not support all of the data types that the MIB objects can be defined as. The SMI document [5] specifies a number of data types which include: counter and gauge. A counter is defined as a non-negative integer which monotonically increases until it reaches a maximum value, then it wraps around and starts increasing from zero. The SMI specifies a maximum value of 2**32-1. An example would be the ICMP statistics structure in the UNIX kernel. The type of each field is an integer and the MIB RFC specifies that these ICMP variables are counters. The ICMP statistics can now possibly wrap around and become negative numbers. A gauge is defined as a non-negative integer, which may increase or decrease, but latches at a maximum value. The SMI specifies a maximum value of 2**32-1.

The most inefficient part of the agent is the */dev/kmem* interface into the kernel. A more efficient interface into the kernel is needed in order for UNIX to fully take advantage of the SNMP philosophy. For instance, direct access to kernel memory would be desirable as opposed to the file-like seeking and reading which must be performed now.

As was mentioned earlier, the SNMP agent communicates with other daemons using IPC to retrieve information. Another viable option to communicate with daemons is by using shared memory. It is not clear whether using shared memory will be more efficient than the method being used in this implementation. It is clear that this option should be explored.

10. The Future

The SNMP is a solution to most of the network management needs of today's Internet. The common information base [4,5] shared with network management efforts in the ISO/OSI domain will provide constructive input based on operational experience to these ISO efforts. By design, the SNMP is a protocol which incurs little overhead on managed network entities, resulting in minimal degradation in the performance of the network component. It is expected that the SNMP will dominate network management for the lifetime of the TCP/IP Internet.

In the past, work has focused on the development of the SNMP protocol and an information base for network management. Implementation efforts in the past have produced an agent and network management station conformant to the SNMP. In the short-term, development efforts will center on enhancements to the user interface of the network management station with the goal of providing a more intuitive network management station.

Experience has shown that the voluminous amounts of data collected through the SNMP will rapidly overwhelm the network manager if such data is stored by means of UNIX flat files. Hence, work will also be performed to provide a means to integrate

such data into an existing database system.

It is clear that the SNMP's greatest shortcoming is the lack of peer authentication facilities. The lack of a coherent method for authentication is a major obstacle to performing network alterations in the SNMP. Plans call for the integration of authentication into the implementation.

11. References

[1] J.D. Case, J.R. Davin, M.S. Fedor, M.L. Schoffstall, *Simple Network Management Protocol*, Request for Comments 1067, Network Information Center, SRI International, Menlo Park, California, September, 1988.

[2] J.D. Case, J.R. Davin, M.S. Fedor, M.L. Schoffstall, *Simple Gateway Monitoring Protocol*, Request for Comments 1028, Network Information Center, SRI International, Menlo Park, California, November, 1987.

[3] V. Cerf, *IAB Recommendations for the Development of Internet Network Management Standards*, Request for Comments 1052, Network Information Center, SRI International, Menlo Park, California, April, 1988.

[4] K. McCloghrie, M.T. Rose, *Management Information Base*, Request for Comments 1066, Network Information Center, SRI International, Menlo Park, California, September, 1988.

[5] M.T. Rose, K. McCloghrie, *Structure of Management Information*, Request for Comments 1065, Network Information Center, SRI International, Menlo Park, California, September, 1988.

[6] J. Postel, *User Datagram Protocol*, Request for Comments 768, Network Information Center, SRI International, Menlo Park, California, August, 1980.

[7] J. Postel, *Transmission Control Protocol*, Request for Comments 793, Network Information Center, SRI International, Menlo Park, California, September, 1981.

[8] International Standards Organization, *Information Processing Systems - Open Systems Interconnection - Specification for Abstract Syntax Notation One (ASN.1) International Standard 8824*, December, 1987.

[9] International Standards Organization, *Information Processing Systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1) International Standard 8825*, November, 1987.

[10] J.D. Case, J.R. Davin, M.S. Fedor, M.L. Schoffstall, "Introduction to the Simple Gateway Monitoring Protocol", *IEEE Network*, Vol. 2, No. 2, pp. 43-49, March 2, 1988.

[11] W. Yeong, M.S. Fedor, M.L. Schoffstall, *SNMP Network Management Station and Agent/Server Implementation*, Version 3.0, NYSERNet Incorporated, September, 1988.

[12] M.S. Fedor, "GATED: A Multi-Routing Protocol Daemon for UNIX", *Proceedings of the 1988 Summer USENIX conference*, San Francisco, California, June 20-24, 1988.

[13] P. Mockapetris, *Domain Names - Concepts and Facilities*, Request for Comments 1034, Network Information Center, SRI International, Menlo Park, California, November, 1987.

[14] P. Mockapetris, *Domain Names - Implementations and Specifications*, Request for Comments 1035, Network Information Center, SRI International, Menlo Park, California, November, 1987.

[15] Neigus, *File Transfer Protocol*, Request for Comments 541, Network Information Center, SRI International, Menlo Park, California, July, 1973.

Limiting Factors in the Performance of the Slow-start TCP Algorithms

Allison Mankin and Kevin Thompson

The MITRE Corporation
Networking Center
McLean, Virginia
703-883-7907

ABSTRACT

Jacobson and Karels' Slow-start TCP offers an effective approach to Internet congestion control. Limiting factors for the Slow-start algorithms were explored by measuring the performance of the Berkeley 4.3+ Slow-start implementation in the presence of increasing gateway load. Two network performance measurement tools, *tcptrace* and *NETMON/iptrace*, were developed for this purpose, and are available in source-code form from the authors. The measurements of Slow-start TCP illustrate both the adaptive range of the algorithms, and also some limitations of the Slow-start technique of estimating gateway congestion.

Background

Congestion is a fundamental problem for the TCP/IP Internet. Traffic from heterogeneous networks, with speeds ranging from 9.6kbps to 1.54Mbps, must share the resources of the IP gateways that form the essential linkages of the Internet. Data is often transferred between hosts on high-speed networks via gateways and low-speed networks.

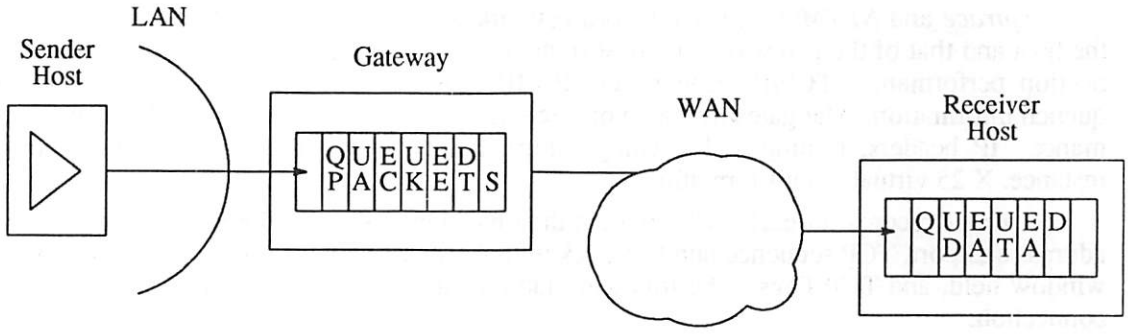


Fig. 1: Receiver Window Flow Control

TCP, designed before the advent of high-speed networks, has a window flow control mechanism keyed to the receiver's buffer space. Overflows of buffers or overloading of resources in the intermediate gateways are not covered by TCP flow control. Figure 1 illustrates this situation. The gateways' only input control mechanism, the ICMP (Internet Control Message

Protocol) source quench message, is a choke packet with at best advisory force. The gateways' input limiting mechanism is to discard packets. Unfortunately, in many TCP implementations, the reliable retransmission mechanism distinguishes poorly between the discarded packets and packets that are just delayed. Responding to delays as though they indicate loss, many TCPs add multiple copies of the same data to the net in response to existing congestion.

The predisposition of the Internet to troublesome congestion and occasional "congestion collapse" was pointed out by Nagle (1984) a year after the TCP standard was published. Recently, Van Jacobson (Lawrence Berkeley Labs) and Mike Karels (Berkeley) accomplished a breakthrough in TCP congestion control, by the design and development of Slow-start TCP (Jacobson, 1988). Slow-start superimposes dynamic window control by the sender on TCP's existing receiver flow control mechanism. It also succeeds in "making the retransmit timer work," through a combination of accurate sampling and a mean deviation filter.

Slow-start TCP has precedents. Bertsekas and Gallager (1987) recommend an adaptive transport send window for highly dynamic environments (such as the Internet). They recognize that it is dependent on an accurate timeout algorithm. Jain (1986) develops and simulates a dynamic window congestion control algorithm called CUTE. This uses transport retransmission timeouts to trigger window reduction, since the timeout is taken as signalling the overflow of gateway capacity. Slow-start uses retransmission timeout in much the same way as CUTE. Ramakrishnan and Jain (1988) refine CUTE with algorithms in which connections receive a one-bit congestion signal from the gateway, instead of depending on the timeout. In introducing BSD 4.3+, Jacobson and Karels verified experimentally (Jacobson 1987) many of Ramakrishnan and Jain's simulation results (the increase and decrease algorithms, fairness), while holding to the timeout signal as sufficient.

Tools

This paper presents performance measurements of the Berkeley UNIX† Slow-start TCP implementation. The measurements are made using tools we have developed at MITRE, *tcptrace* and *NETMON/iptrace*. Both are traffic monitors, programs which efficiently yield a detailed, timestamped log of network packets. *tcptrace* logs TCP connection traffic as it passes through an Ethernet. *NETMON/iptrace* logs the traffic forwarded through an IP gateway. *tcptrace* is similar to Van Jacobson's *tcpdump* program available for SunOS* We added logging of source quenches. *NETMON* is a portable kernel instrumentation facility we developed for Berkeley UNIX systems. It is analogous to SunOS NIT(Network Interface Tap), but it can be used to record network traffic on any network interface, not only the Ethernet. This versatility makes possible our gateway monitoring. It has also allowed us to run *tcptrace* on varied Berkeley UNIX-based systems. Both performance measurement tools are available in C source-code form on request.

tcptrace and *NETMON/iptrace* record network activity in two frames of reference, that of the host and that of the gateway. The host frame of reference includes data relevant to TCP connection performance: TCP/IP headers and ICMP (Internet Control Message Protocol) source quench information. The gateway frame of reference includes data relevant to IP gateway performance: IP headers, routing and queuing information, and network interface indications, for instance, X.25 virtual circuit formation.

tcptrace records for each TCP segment the time-stamp, source address and port, destination address and port, TCP sequence number, packet total length, TCP acknowledgment number, TCP window field, and TCP flags. The following table is an example of *tcptrace* output tracing one connection:

† UNIX is a trademark of A T & T Bell Laboratories.

* SunOS and NIT are trademarks of Sun Microsystems, Incorporated.

Table 1: Example of tcptrace Output

```
10:29:59.88  bede.mitre.org.1105 > CS.ROCHESTER.EDU.9 seq# 0
len 40 ack# 0 win: 4096     S       NEW
-
•••
-
10:30:08.60  bede.mitre.org.1105 > CS.ROCHESTER.EDU.9 seq# 3072
len 552 ack# 0 win: 4096 A
-
10:30:08.62  bede.mitre.org.1105 > CS.ROCHESTER.EDU.9 seq# 3584
len 552 ack# 0 win: 4096 A
-
Source Quench by 128.29.1.7 for -> 128.29.1.2.1105 > 10.0.0.15.9 seq# 3584
-
•••
-
10:30:13.74  bede.mitre.org.1105 > CS.ROCHESTER.EDU.9 seq# 3584
len 552 ack# 0 win: 4096 A
-
•••
-
10:32:31.82 CS.ROCHESTER.EDU.9 > bede.mitre.org.1105 seq# 1
len 40 ack# 40961 win: 4096 A F        CLOSE
-
```

The values of the TCP sequence number (seq#) and acknowledgement number (ack#) are relative to the start of the connection, as in *tcpdump*. Connection establishment and closing are identified with easy-to-filter labels. Source quenches are recorded with a clear label, the address of the issuing gateway, the quenched segment's source and destination address, and its sequence number.

NETMON is an independent kernel subsystem, in that it does not depend on any vendor- or hardware- specific portion of the BSD UNIX operating system. The user-level interface accessed by *iptrace* is a special file. *NETMON* is integrated into a BSD UNIX kernel by adding a few invocations of a function (''probes'') to the input and output routines of each protocol implemented in the kernel. Incidentally, *NETMON* is protocol-independent, having no dependencies on the particulars of any protocol. In our gateway, IP, Ethernet and X.25 have *NETMON* probes inserted in them. It would be easy to integrate *NETMON* into other protocols as well, if they were present. For instance, to measure the performance of an OSI gateway, we will insert probes in the ISO Connectionless Network Service (CLNS) input and output routines.

NETMON/iptrace gives an internal view of protocol performance. The picture of delayed and dropped traffic is detailed by recording the following information: packet id, time-stamp, protocol location/event code, queue length, queue id/interface id, packet total length, source address and port, and destination address and port. Table 2 is an example of the output of *iptrace* in logging *NETMON* information tracing a single packet being forwarded through the gateway.

Table 2: Example of NETMON/iptrace Output

```
80202B00 06:57:35.57 ethIn>0 0 len -1:
-
80202B00 06:57:35.57 ipInForw>0 qe0 len 552: 128.29.1.12.1136 > 10.0.0.78.9
-
80202B00 06:57:35.58 ipOutForw>-1 dda0 len 552: 128.29.1.12.1136 > 10.0.0.78.9
-
80202B00 06:57:35.58 x25Out>1 5 len -1:
-
•••
-
80202B00 06:57:36.76 x25OutDeq>2 5 len -1:
-
```

The packet id is the address of the first mbuf containing the packet; in BSD UNIX, this address remains constant as the packet is passed from layer to layer (Leffler et al, 1986). The interspersed records of one packet are matched using this field. In the example output, there were ten records between the X25Out and the X25OutDeq lines.

The location (e.g. ipIn—*iptrace* concatenates this field and the next) identifies the protocol layer the packet is passing through. For each protocol location, there are various event codes (e.g. Forw) corresponding to probe points. There is one event indicating the normal condition (i.e. packet successfully received or sent). Then there is one or more indicating other conditions, for example, packet dropped due to queue overflow, or packet held pending address resolution.

The queue id/interface id field identifies multiple queues, such as those of X.25. In the example, the packet is enqueued on the output queue associated with X.25 LCN (Local Connection Name) 5. The same field in the IP-level records identifies the network interface on which the packet arrives and departs. This is especially useful in a gateway which links multiple networks of one type.

NETMON's queue information (the queue length is the number following the angle bracket) is very interesting for congestion control evaluation. The times at which packets enter and leave queues in the gateway and the length of the queues directly indicate the level of congestion present. In the example, the packet finds one packet ahead of it on the X.25 output queue when it is enqueued. It remains on the queue for 1.18 seconds (06:57:36.76 - 06:57:35.58). By the time that it is dequeued, two other packets have been enqueued behind it. A value of -1 in the queue length field indicates that there is no queuing or dequeuing at a given probe point. The length field is also of importance for performance instrumentation. The -1 value is used when the packet's length cannot be conveniently recorded.

The example illustrates data obtained from *NETMON's* measurement mode. It does not include full header information. In header mode, the records for ipInForw and ipOutForw would both have a complete copy of the packet's IP header appended. This would be valuable for tracing the IP header changes made by the gateway. Header mode adds some byte-copying and buffering overhead that is undesirable for performance measurement; however, it is useful for debugging (e.g. capturing ill-formed packets, or "bogons").

An important question about an internal monitoring facility such as *NETMON* is how much it affects the network performance of the system it resides in. A moderate amount of added overhead is acceptable because the *NETMON* event information cannot be obtained in other ways. We conducted numerous tests to be sure that less than a few percent of packet delays and drops recorded by *NETMON* could have been caused by *NETMON*. The main approach to these tests was to independently measure traffic sets with *NETMON* disabled and enabled in close succession.

Performance of Slow-start TCP

The BSD 4.3+ Slow-start implementation was made publicly available in April 1988. During the summer of 1988, we conducted a set of experiments with this code to explore limiting factors of the dynamic windowing algorithms. The guiding questions were: with multiple connections to cause congestion at the gateway, do the connections continue to perform dynamic window control efficiently? This is to say, do they decrease their windows enough to reflect the availability of only a small share of gateway capacity? As delays increase, do the timer estimation algorithms continue to work well and avoid spurious retransmissions? Are some connections eventually denied service (which might be desirable)?

Because a Slow-start connection increases the size of its dynamic send window until a retransmission timeout occurs, Slow-start requires some gateway queue overflows. The connection must test the limit of the path bandwidth, otherwise bandwidth that becomes free will not be detected and used. A last question in our Limiting Factor experiments was whether at some point the multiple connections would increase their flow over the limit so often that queue overflows would be continual.

The experiments were carried out in the MITRE Internet Testbed. This was an Ethernet with a gateway to the ARPANET, and was largely devoted to performance testing. Slow-start TCP was installed in four hosts (SUN 2's and DEC MicroVAXes**). The gateway was a MicroVAX running BSD 4.3+, interfaced to the ARPANET with an Advanced Communications Corporation ACP 5250 Standard X.25 board. Successively higher numbers of bulk-transfer connections were run simultaneously using various source-destination matrices (the data transfer was from memory on the local hosts to the RFC 863 Discard Protocol on the remote hosts). Traces of the connections were collected as the segments crossed the Ethernet using *tcptrace*. Connection metrics including throughput, drop rate, and throughput fairness were determined. For some cases, extra experiments were run to collect gateway queue lengths using *NETMON/iptrace*.

Observations

The first set of experiments, the Different Destinations set, provided a baseline measurement of multiple Slow-start connections. Each connection was to a different ARPANET host. No more than four connections originated from each testbed host. The first run involved concurrent transfer of 40 kilobytes over two connections. The second run used three connections, and so on, up to 16 connections. All traffic hosts used had similar processing capacity and all had negligible CPU loads, except for traffic generation, during the experiment.

At the conclusion of the runs, several performance metrics were derived. Retransmissions were counted and summed for each run. In this set of experiments, no source quenches appeared in the trace data. Therefore all of the retransmissions shown in Table 3 were the result of timer under-estimation rather than packet loss. The numbers were low (consider that each connection had a minimum of eighty segments to send), and they occurred for the most part during the early, start-up phase of the connections.

Table 3: Retransmissions—Different Destinations

Conns	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Rxts	6	6	5	9	9	12	15	15	16	3	21	16	2	20	22

The throughput of each connection was computed by calculating the bytes per second in the interval between the last establishment of a connection and the first closing of a connection. Net

** DEC and MicroVAX are trademarks of Digital Equipment Corporation.

throughput included only bytes that were not retransmissions. The summed net throughput for different destinations increased nearly additively with the number of connections. As illustrated in Figure 2, only with thirteen or more connections did the total begin to level, and then perhaps decline. The peak total throughput, at 13 connections, corresponded to over 70% of the nominal link bandwidth of 56 Kbps. The upper line in Figure 2 was the summed gross throughput of the connections, that is, the total bytes transmitted including retransmits; as already discussed, the number of retransmits was low.

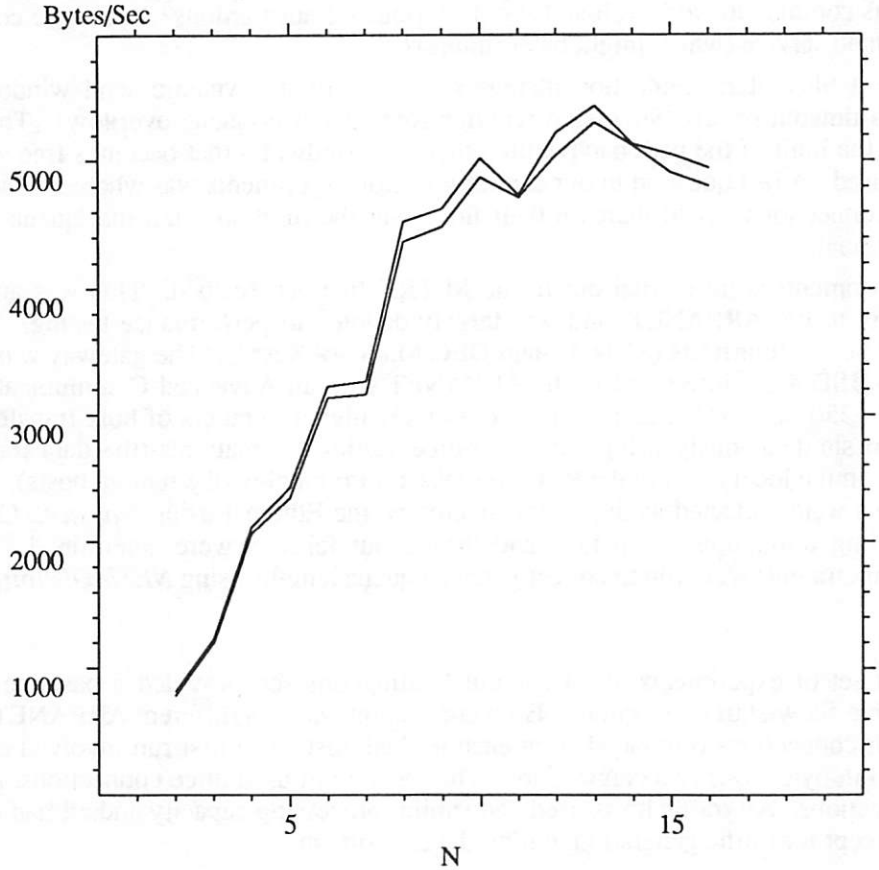


Fig. 2: Summed Throughput of N Connections (Different Destinations)

The second experiment set, the Same Destination set, tested Slow-start TCP for a threshold on the number of competing connections beyond which throughput declined. All connections were to the same ARPANET host. In addition to the destination host resources, these connections were competing for the resources of a single ARPANET virtual circuit. This experiment set followed a similar procedure to the previous one, except that fewer runs were performed (the runs were with three, four, five, six, twelve, and sixteen connections).

The summed net throughput as the number of connections increased was observed to be fairly constant, as seen in Figure 3. This indicated that the available shared resources (almost certainly, the ARPANET virtual circuit resource) remained constant through the experiment runs.

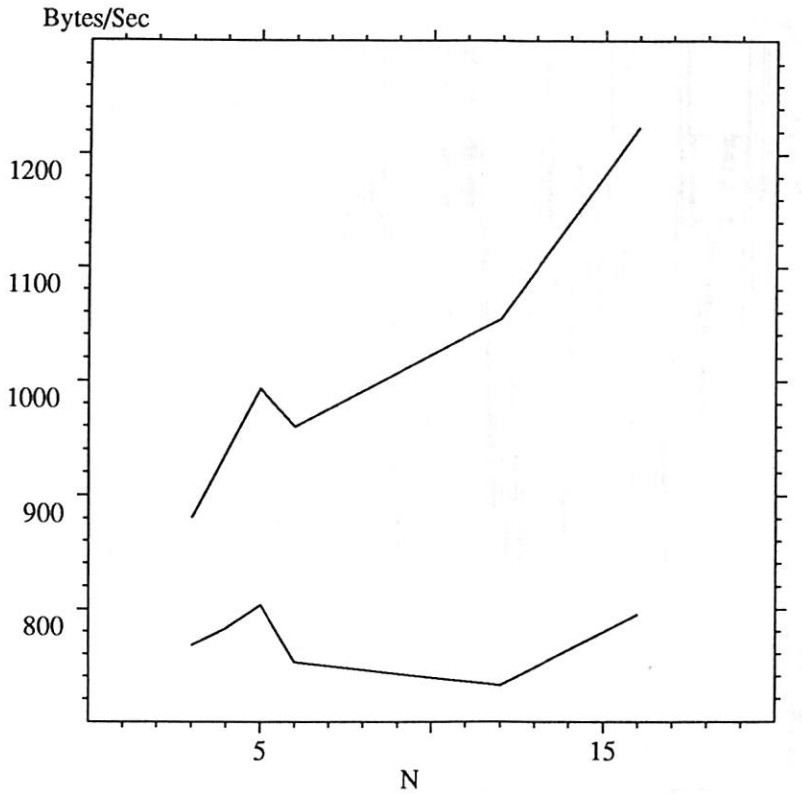


Fig. 3: Summed Throughput of N Connections (Same Destination)

The summed retransmissions of each run in this set were very different from the Different Destination set. While the number of spurious retransmissions was comparably low, the number of dropped packets became significant. These statistics are summarized in Table 4.

Table 4: Retransmissions—Same Destination

Connections	Retransmits	Drops	Spurious
3	33	20	13
4	42	36	6
5	81	68	13
6	101	86	15
12	322	314	8
16	337	328	9

Since most retransmissions were due to packet drops, the difference between the upper and lower lines in Figure 3 became a measure of the packets dropped per second, Going from the initial run, with three connections, to the final runs, with twelve and sixteen connections, the rate of this overflow increased from about 100 bytes per second (about 0.2 packets) to about 500 bytes per second (1 packet). This change in drop rate could be viewed as conservative; in severe congestion, the twelve connections were collectively overestimating the bandwidth by less than a factor of two. However, the sharp upturn of the upper line at sixteen may suggest that Slow-start's behavior would be less conservative if the congested conditions got any worse.

Gateway queue behavior was measured for some of the Same Destination cases. As Figures 4 and 5 show, the frequency of overflows and the congestion of the gateway's ARPANET output queue could easily be seen for different numbers of Slow-start connections. Figure 4 shows that as fews as three connections could keep the queue between full and overflowing for steady periods of time.

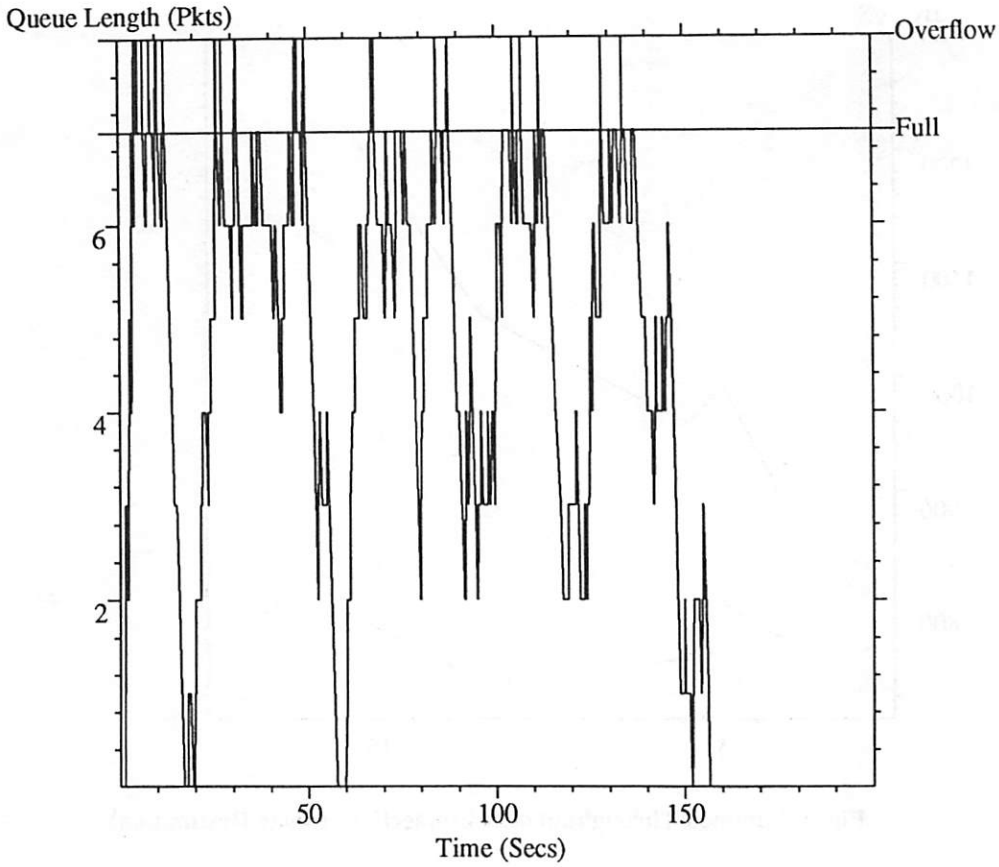


Fig. 4: Queue Behavior of 3 Conns (Same Dest)

Finally, with regard to denial of service in the Same Destination set, we planned several trials to measure more than sixteen simultaneous connections, but, in these, some connections timed out before they could become established. This proved to be due to a fixed-length (non-adaptive) connection establishment timer of 75 seconds in BSD 4.3+ TCP, an overlooked inheritance from the pre-Slow-start BSD TCP. In no runs did a connection time out once it was established. This was despite numerous 100 second or more delays before acknowledgement of a segment (in the case of segments dropped multiple times) in the twelve and sixteen connection runs.

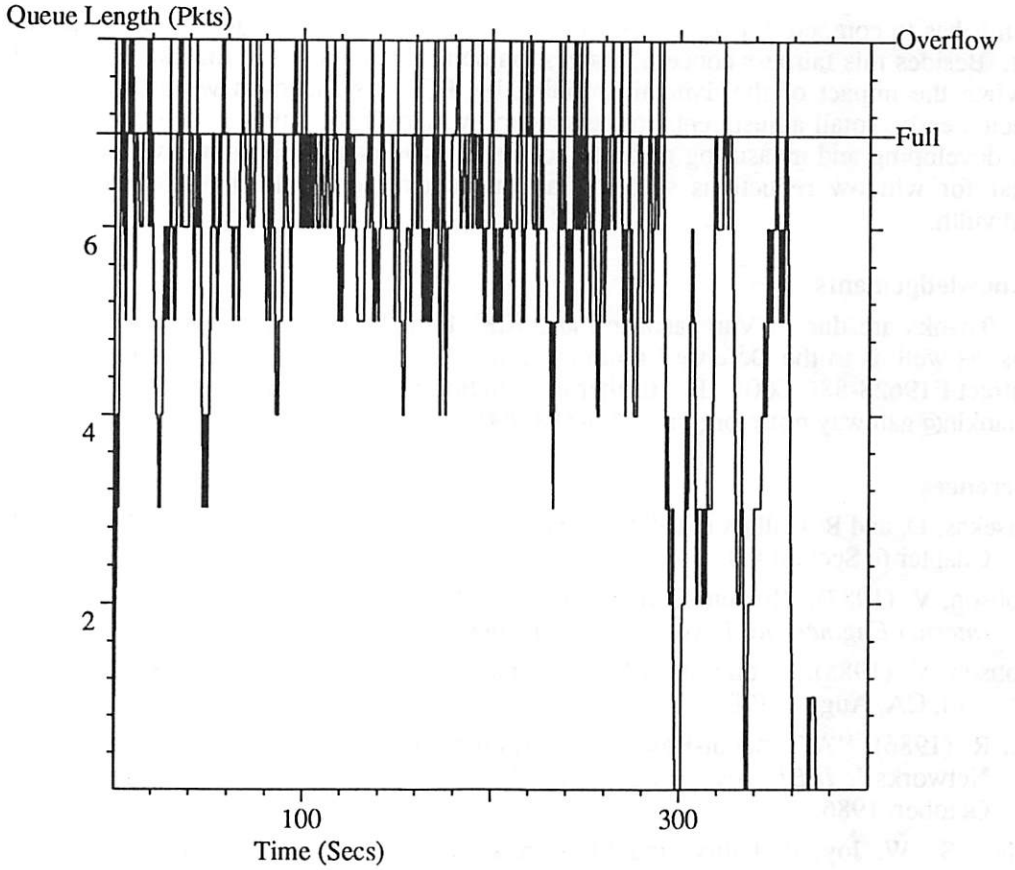


Fig. 5: Queue Behavior of 6 Conns (Same Dest)

Conclusions

The data presented here is still being analyzed and more data, from more network configurations, must be collected to supplement it. The goal of this paper has been to illustrate aspects of the limiting factors in the performance of Slow-start TCP, along the way showing the use of the *tcptrace* and *NETMON* tools.

The Same Destination experiment set subjected Slow-start TCP to conditions of extreme stress. Under those conditions, the timeout estimation algorithm was very robust, over a wide range of delays in a large sample. TCP's tendency to inject multiple copies of data into the network during congestion is cured in this implementation. Once established, all connections completed their data transmission. This success may have a cost, since it may be hard for the network to recover from congestion if connections do not give up and if new connections continue to gain access.

The conditions seen in the experiment (up to the case of six) occur more often in the real networks than it would at first appear. Multiple concurrent TCP connections through one pair of ARPANET gateways (their endpoints not the same) result in this scenario. Tuning by use of the recent change in ARPANET software allowing multiple virtual circuits, and thus multiple gateway queues, between popular endpoints, should be recommended. But the observations of this ARPANET situation lead to concerns beyond tuning. Some resource will always be competed for. Although the Slow-start algorithms greatly improve on the congestion control options originally available to the TCP-IP Internet, the probe of the dynamic window appears to lead to significant loading of the bottleneck.

Jacobson (1988) suggests that the gateways need to implement more powerful congestion control than packet discard and source quench messages, because Slow-start slows too much

when it has to compete for resources with TCP implementations which do not implement Slow-start. Besides this fairness concern, congestion control techniques in the gateway may be able to alleviate the impact of the dynamic probing by TCP. As Jacobson writes, "If [congestion is] detected early, small adjustments to the senders' windows will cure it." In ongoing work, we are now developing and measuring gateway congestion avoidance algorithms which may be able to signal for window reductions sooner than queue overflow but still offer fair allocations of bandwidth.

Acknowledgements

Thanks are due to Van Jacobson and K.K. Ramakrishnan for several educational discussions, as well as to the Defense Communications Agency for solely supporting this work under Contract F19628-88C-0001. For further information about our software, contact Allison Mankin at mankin@gateway.mitre.org or at 703-883-7907.

References

Bertsekas, D. and R. Gallager (1987), *Data Networks*, Englewood Cliffs, NJ: Prentice Hall, 1987. Chapter 6, Section 4.3.

Jacobson, V. (1987), "Recent Congestion Control Efforts for 4.2/4.3 BSD", *Proceedings of the Internet Engineering Task Force, November 1987*, IETF-87/4P, SRI-NIC.

Jacobson, V. (1988), "Congestion Control and Avoidance", *Proc. ACM SIGCOMM '88*, Stanford, CA, August, 1988.

Jain, R. (1986), "A Timeout-Based Congestion Control Scheme for Window Flow-Controlled Networks", *IEEE Jour. on Selected Areas in Communications*, Volume SAC-4 No. 7, October, 1986.

Leffler, S., W. Joy, R. Fabry, and M. Karels, (1986), "Networking Implementation Notes—4.3BSD Edition", Berkeley, CA: Computer Science Division, University of California, Berkeley, 1986.

Nagle, J. (1984), "Congestion Control in IP/TCP Internetworks", RFC-896, SRI-NIC.

Ramakrishnan, K.K. and R. Jain, (1988), "A Binary Feedback Scheme for Congestion Avoidance in Computer Networks with a Connectionless Network Layer", *Proc. ACM SIGCOMM '88*, Stanford, CA, August, 1988.

The Shared Memory Server

Alessandro Forin, Joseph Barrera, Richard Sanzi

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

This paper describes the design and performance evaluation of a virtual shared memory server for the Mach operating system, providing an extension to Unix to support distributed shared memory. In this respect, it subsumes standard facilities like the Unix System V shared memory facility. The server runs in user-mode and provides sharing of read/write memory between processes, regardless of their machine-location. A number of memory coherency algorithms have been implemented and evaluated, including a new distributed algorithm that is shown to outperform centralized ones. Some of the novel features of the server include support for machines with multiple page sizes and for heterogeneous processors. Performance measurements of the server and of some applications are presented, and the intrinsic costs evaluated.

1. Introduction

Shared memory multiprocessors are becoming increasingly available, and with them a faster way to program applications and system services via the use of shared memory. Currently, the major limitation in using shared memory is that it is not extensible network-wise and therefore is not suited for building distributed applications and services. Example uses of a distributed shared memory facility include file systems, process migration, databases, parallel languages like Ada or Multilisp, and systems for parallel and distributed programming [11, 2, 10]. More motivation for a distributed shared memory facility comes from the increasing interest that hardware designers show in non-shared memory multiprocessors: the Nectar project [1] at CMU for instance uses fast fiber optic links. This will reduce the end-to-end time to send a 1 kilobyte message from the tens of milliseconds range of the current ethernet to the tens of microseconds range of the fiber.

The Mach virtual memory system allows the user to create memory objects that are managed by user-defined processes, called *external pagers* in [12]. An external pager is a process responsible for providing data in response to page faults (pagein) and backing storage for page cleaning (page-out) requests. This is precisely the function of the in-kernel disk pager. The only difference is that the user-specified pager process can manage the data in more creative ways than the designer of the in-kernel pager may have envisioned. This paper describes the design and performance evaluation of one such memory server, which provides shared memory semantics for the objects it manages. The server provides unrestricted sharing of read/write memory between processes running either on the same machine or on different machines. In the first case, all processors have direct access to common physical memory (architectures with Uniform Memory Access time (UMA) or Non-Uniform Memory Access time (NUMA)) and the server provides a flexible management of shared memory. In the second case, processors do not have any way to access common physical memory (architectures with No Remote Memory

Access (NORMA)) and the server provides it as virtual shared memory, migrating and replicating virtual memory pages between processors as needed.

To understand the properties of a distributed memory facility the performance characteristics of the server itself and of various application programs have been evaluated. To measure the effects of different page management policies, a number of memory scheduling algorithms have been implemented and evaluated, including a new distributed algorithm that outperforms centralized ones. Some of the features of the algorithms described include support for machines with different page sizes and for heterogeneous processors. The algorithms service page faults on multiple machines by migrating read-write pages, replicating read-only pages, scheduling conflicting or overlapping requests appropriately and tagging and translating memory pages across incompatible processors. The experiments with application programs were designed under the assumption that the amount of information that is exchanged in each synchronization step is the key factor. Applications at the extreme of the spectrum have been analyzed in detail.

2. Shared Memory Within a Machine

The first goal of the server is to provide sharing of read/write memory between processes running on the same machine. This overcomes the constraint of the standard Mach memory inheritance mechanism that the shared memory must have been allocated by some common ancestor, as well as a security check in the implementation of the Unix exec(2) system call that deallocates all of the process's address space. The server provides the user with a call to create *memory objects*, logical pieces of memory that are identified by ports. A memory object can be used by a process in a call to the *vm_map()* kernel primitive, which maps some portion of the object into the process's address space at some virtual address. Note that since a port can only be transmitted in a message, memory objects are entities protected by the kernel. Note also that access to ports can be transmitted over the network, and therefore the *vm_map()* primitive allows for networked shared memory.

The process can access the memory normally, and the kernel delegates the paging duties to the user-level memory manager (external pager) that is responsible for the memory object. This is done via an asynchronous message interface between the pager and the kernel which is described in more detail in [12]. The external pager interface allows pagers to control the managing of main memory by the kernel, so that main memory effectively acts as a common cache for memory objects. The various operations therefore have the flavor of cache control functions: when a process first accesses a page it takes a page fault, and the kernel sends to the pager a *memory_object_data_request()* message to request the missing page, just like in a cache miss. The server provides the page in a *memory_object_data_provided()* message. Other messages allow a pager to request a page flush or specify the caching and copy policies for the object. Figure 2-1 informally lists the messages and procedures defined by the external pager interface and by the shared memory server.

From user to pager

create_memory_object(initial size)
RPC, Creates a new memory object and returns the associated port.

memory_object_replicate(object)
RPC, When using the distributed pager, create a local copy of the memory object.

memory_object_tag(tag, page range)
RPC, When using heterogeneous processors, assign a type tag to a portion of the memory object.

From user to kernel

vm_map(task, memory_object, address range, attributes)
RPC, Maps an object into a task's address space.

vm_deallocate(task, address range)
RPC, Removes all mappings for the given address range.

From kernel to server

memory_object_init(pager, control_port)
MSG, Contact the pager of an object which is mapped for the first time, for initial handshake.

memory_object_data_request(page range, protection)
MSG, Request for a (range of) page which the kernel does not have in its cache.

memory_object_data_unlock(page range, protection)
MSG, Requires more access permissions for a page.

memory_object_data_write(page range, pages)
MSG, Pageout of dirty pages from main memory.

memory_object_lock_completed(page range)
MSG, Completion of the requested paging operation.

memory_object_terminate()
Notification of removal from cache.

From server to kernel

memory_object_set_attributes(attributes)
MSG, Confirms availability completing initial handshake, specifies initial attributes.

memory_object_data_provided(page range, pages)
MSG, Provides page(s) data to the cache.

memory_object_data_unavailable(page range)
MSG, Zero fill page(s).

memory_object_lock_request(object, request, reply_port)
MSG, Cache control request, e.g. page flush or granting of write access.

Figure 2-1: Summary Of The External Pager Interface

3. A Simple Algorithm

The shared memory server has been structured in an object-oriented fashion, so that it is possible to have memory objects with different behaviors. When a memory object is mapped by processes on multiple machines, the pager needs to manage multiple copies of memory pages in some coherent way. The various management policies for objects are provided by different implementations of a set of operations: an implementation is called *fault scheduler* in the following, because the goal of the module is to schedule read and write faults on different kernels in the best way, just like ordinary schedulers schedule the execution order of various processes. One of the many reasons for this choice is to allow experimentation with various algorithms and heuristics. At object creation time, a user can choose which specific scheduling policy will be applied to the new object, or rely on the default one. All the algorithms we describe maintain *strict memory coherence* on the objects they manage, e.g. there is no stale data because at any given time there is only one version of a page.

This Section describes a very simple scheduler that provides centralized, single page-size objects. There is only one pager process for each memory object, but different objects might be allocated to different pager processes to reduce service contention. Since Mach IPC is location transparent, the location of the pager process is also transparent to the client kernels. A later Section will describe how this algorithm is modified to allow distributed, coordinated management of a single object between separate pagers on different machines. Ownership of a page is transferred among kernels as needed: the *owner* of the page is the kernel that currently has write access to the page. When no kernel has write access to a page the scheduler itself is the owner, and multiple kernels are allowed to have read-only copies of the page. The simple scheduler's algorithm is an automaton with four per-page states, which correspond to the four conditions in which a page can be:

- **Read**: There are no writers, there may be readers with a copy, the server has a valid copy. This is the initial state.
- **Write**: There is one writer, there are no readers and no one is queued waiting, the server does not have a valid copy.
- **ReadWait**: There is one writer, some readers are waiting, the server does not have a valid copy and has asked the current owner to return the page to the server.
- **WriteWait**: There is one writer, some writers are queued waiting, there may be readers waiting, the server does not have a valid copy and has asked the current owner to return the page to the server.

Transitions between states are driven by the requests that are made by client kernels. In practice, not all requests make sense in all states. For example, a kernel will not pageout a page that has not been modified. The server accepts four input message types (requests), which the scheduler handles in three procedures:

- *read_fault()*: a kernel requests read access to a page (memory_object_data_request).
- *write_fault()*: a kernel requests write access to a page and either needs a fresh copy of the page (memory_object_data_request) or does not (memory_object_data_unlock).
- *pageout()*: a kernel flushes out the page to the server (memory_object_data_write and memory_object_lock_completed).

These three functions do all the necessary work. A pseudo code description of how they operate on a page appears in Figures 3-1, 3-2 and 3-3. It can be assumed that all procedures keep the page locked and that messages are processed in the order of arrival. This pseudo code will be used again later to describe the distributed algorithm. The remaining procedures are either for initialization, termination, or recovery from kernel crashes. The pseudo code indicates that writers are queued in FIFO order, while readers do not need to be ordered. Writers take precedence over readers. Other, possibly more complicated policies might be needed, for instance to deal with multiple page sizes, or to avoid reader starvation.

An example will help clarify the following discussion. Since all the processes on one machine use the same copy of the memory object's pages (cache copy, possibly mapped into the various address spaces with different protections), we can pretend there is a single process per machine. Let us assume that the process makes a read access to a page. The page is not in the cache,

```
read_fault(page, kernel)
    switch ( page->state ) {
    case Read:
        memory_object_data_provided(kernel)
        break
    case Write:
        page->state = ReadWait
        memory_object_lock_request(page->owner, FLUSH(page), owner_self)
        break
    default: /* just enqueue */
    }
    set_add(page->readers, kernel)
```

Figure 3-1: Handling of Read Faults

hence the kernel sends a *memory_object_data_request()* message to the pager. If the page is in Read state (the initial state), the server immediately sends the page in a *memory_object_data_provided()* message, with read-only protection. If the process makes a subsequent write access, the kernel sends a *memory_object_data_unlock()* message to request a protection upgrade which will be granted in a *memory_object_lock_request()* message, unless the page has changed state in the meantime. If the page is not in Read state, the kernel's request is enqueued and possibly the current writer is asked to page out the page via a *memory_object_lock_request()* message. When the page is actually paged out, the *pageout* procedure dequeues the next write access request and satisfies it, or satisfies all read requests at once.

```
write_fault(page, kernel)
    switch ( page->state ) {
    case Read:
        set_remove( page->readers, kernel)
        forall( readers )
(1)         memory_object_lock_request( reader, FLUSH(page), owner_self )
        page->readers = empty_set
(2)
        page->state = Write
        page->owner = kernel
        if (needs_data)
            memory_object_data_provided( page->owner )
        else
            memory_object_data_unlock( page->owner )
        break
    case Write:
        memory_object_lock_request( page->owner, FLUSH(page), owner_self )
        /* fall through */
    case WriteWait:
    case ReadWait:
        page->state = WriteWait
        enqueue( kernel, page->writers )
    }
```

Figure 3-2: Handling of Write Faults

3.1. Multiple Page Sizes

The simple scheduler described above can only be used by machines with the same page size, an unpleasant restriction. Moreover, in Mach the size of a virtual page can be changed and set even on a per-machine basis. Transforming a single page size scheduler into a multiple page size scheduler is not trivial. Our multiple page size scheduler solves the problem by two means:

- for requests *smaller* than the scheduler page size, the request is rounded up to the

```
pageout(page, kernel, data)
(3) switch( page->state ) {
    case Read:
        return     /* never happens */
    case Write:
        save(data)   /* true pageout */
        page->state = Read
        page->owner = owner_self
        break
    case WriteWait:
(4)
        save(data)
        page->owner = dequeue( page->writers )
        memory_object_data_provided( page->owner)
        if (!page->writers)
            if (page->readers)
                page->state = ReadWait
            else
                page->state = Write
        if (page->readers || page->writers) {
            deschedule_myself()
            memory_object_lock_request( page->owner, FLUSH(page), owner_self)
(5)
        }
        break;
    case ReadWait:
        save(data)
        forall(readers)
            memory_object_data_provided(reader)
        page->state = Read
(6)     page->owner = owner_self
    }
```

Figure 3-3: Handling of Pageout Requests

scheduler page size, and

- for requests *larger* than the scheduler page size, the request is fulfilled by multiple scheduler pages (shipped all at once), after appropriate locking.

Locking is accomplished via a queueing mechanism. Some complications arise from the requirements of avoiding false contention and descheduling of kernels until absolutely necessary, and of satisfying requests as quickly as possible while maintaining fairness. A description of this mechanism can be found in [5].

3.2. Heterogeneous Processors

Parallel programs that use a distributed shared memory facility should not be constrained to run on a uniform set of processors. Such a constraint is undesirable because as the number of machines available at a given site increases one typically observes an increased variation in their types as well. Unfortunately, interfacing heterogeneous processors not only creates the problem of potentially different page sizes, but also raises the issue of different machine representations of data objects. This problem goes beyond the *byte order* problem, since different processors are free to assign any given meaning to any given sequence of bits. A clear example is the case of floating point numbers.

The problem can be separated in two sub-problems: hardware data types (e.g. integers) and software data types (e.g. C records). A general purpose server solves the problems for the first

class of types, and can be extended to cope with the second class of types. Quite simply, our pager assigns a type tag to each segment of a paging object and makes the appropriate translation (if necessary) when sending data from that segment to a kernel. The interface with the application program is defined by the *memory_object_tag_data()* RPC from the client to the pager that assigns a type tag to a segment. This operation is typically used by a dynamic memory allocator to fragment shared memory in typed segments, each segment containing only data of the given type. The standard Unix BSD malloc(2) memory allocator for C was modified to allocate typed data, as exemplified in Figure 3-4. Although different types cannot be mixed in a structure, one can always resort to a level of indirection, building records that only contain pointers to data.

```
extern char
        *tmalloc( type_tag, num_elements )
enum { t_int8, t_int16, t_int32, t_float32, ... } type_tag;
unsigned long int num_elements;

#define malloc_short(n) (short*)tmalloc( t_int16, n)
...
```

Figure 3-4: A Typed *malloc()*

4. A Distributed Algorithm

The motivations for a distributed algorithm are manyfold. A centralized server is a solution that does not scale up. When many kernels share many memory objects serviced by the same pager the availability of each object decreases, because the pager becomes the bottleneck where all requests pile up. Even when few kernels are involved, the location of the server is important because local and remote messages might have very different costs. A distributed solution that can allocate any number of servers on any number of machines is more usable. In this way the sharing of memory between processes located on the same (multi)processor is decoupled from unrelated events on other machines.

The approach is simple: *treat each remote server just like another kernel, and apply the algorithm of the centralized case.* The reader may wish to go back to Figures 3-1, 3-2 and 3-3 and review the algorithm substituting the word "kernel" with "client", which now means either a kernel or (more likely) a fellow server. A pager will now accept a *memory_object_lock_request()* message just like a Mach kernel does and treat it as a fault notification, invoking read_fault() or write_fault() as appropriate. A *memory_object_data_provided()* message is handled by the pageout() procedure.

Note now that the notion of the "owner" that each pager has does not need to be exact at all times. It is quite possible, actually highly desirable, that a pager be able to ask a second pager to transfer a page directly to a third one who needs it, without handling the page directly. We call this optimization **forwarding**, to catch both the positive effect of avoiding one message hop, and the (minor) negative effect of producing a new type of activity: the act of forwarding a mis-directed page fault message to the correct destination. Implementing forwarding requires relatively simple changes to the centralized algorithm.

```
(1)     memory_object_lock_request( reader, FLUSH(page),
            is_server(page->owner) ? kernel : owner_self)

(2)     if (page->owner != owner_self) {
            memory_object_lock_request(page->owner, WRITE_FAULT(page), owner_self)
            enqueue(page->writers, kernel)
            page->state = WriteWait
            return
        }

(3)     if (kernel != page->owner && !hinted(page))
            page->owner = kernel
        hinted(page) = FALSE

(4)     if (!page->writers) {
            page->owner = owner_self
            goto ReadWait
        }

(5)     if (is_server(page->owner))
            page_state = WriteWait        /* pretend */

(6)     if (!is_server(kernel))
            page->owner = owner_self
```

Figure 4-1: Modifications to the Distributed Scheduler to Implement Forwarding of Page Faults

Figures 4-1 and 4-2 illustrate the changes and additions to the pseudo code to implement forwarding. A pager creates a local copy of a memory object when a user asks for it. The initial state of all pages in this case is the *Write* state, and the owner is the pager from which the object has been copied. Of course, no real copy is actually done. Note that it is possible to copy from another copy, and that the pager need not have complete knowledge of all the kernels involved. The handling of read faults does not change. While handling write faults, at line (1) all readers are informed of who the new owner is, if it is a different pager. At line (2), a check is added to see whether the true owner actually is another pager, in which case the fault is queued and the state of the page modified accordingly. In the pageout() procedure at line (3) it is necessary to handle the case where the pager has incorrect information about the true owner. Note that the pager might have received a *hint* about who will eventually become the owner because it forwarded a write fault. At line (5) it is necessary to handle specially the case when a page is given to a server queued for writing, while having other readers waiting. The immediate request to have the page back pretends that there are writers queued anyway, to prevent the race that would otherwise arise. Line (4) jumps to the correct code in case the last writer had actually been serviced. Line (6) handles the fact that if the pager only receives read-only access to the page it does not become the owner of the page.

Two new procedures, described in Figure 4-2, are used to check whether a page fault must be forwarded and to handle invalidations of read-only pages. A *memory_object_lock_request()* message is handled first by the page_fault() procedure, which forwards it if necessary. The fault is definitely not forwarded if the pager has ownership of the page, or the pager has already asked the current owner for write access to the page (state WriteWait), or if the pager has (state Read) or is about to have (state ReadWait) a read-only copy of the page and the fault is a read fault. In other words, a fault is only forwarded to another server when the pager has no current interest in the page whatsoever. An invalidation of a read-only page is generated at lines (1) and (7) if the

reader is a server, and is handled in the invalidate_page() procedure. This is the only new message type needed.

```
        invalidate_page(page, owner)
         if (page->state != Read)
             return     /* sanity check */
         forall (readers)
(7)              memory_object_lock_request(reader, FLUSH(page), owner)
         page->state = Write;
         page->owner = owner;

        page_fault( page, who, fault_type)
         if ((page->owner == owner_self) ||
             !is_server(page->owner) ||
             (page->state == WriteWait) ||
             ((fault_type == READ) && (page->state != Write))) {
             if (fault_type == READ) read_fault(page, who)
             else write_fault(page, who)
             return
         }
         /* Forward */
         send_page_fault(owner,who,page)
         if (fault_type == WRITE) {
             page->owner = who
             hinted(page) = TRUE
         }
```

Figure 4-2: Additions to the Distributed Scheduler to Implement Forwarding of Page Faults

Forwarding creates problems for a closed form analysis, since the effect of forwarding of both page locations (page faults) and invalidations (page flush) are difficult to model. Our claim is that in actual use one will typically see only the two extreme cases: pages that are frequently accessed in write mode by many parties, and pages that are accessed infrequently, most likely in read mode. Even if a page is accessed infrequently, it is hard to generate a faulting sequence that produces many forwarding messages. This claim is supported by the experience with actual application programs. Infrequently accessed pages do not affect performance. The bottlenecks derive very easily from the opposite case. Our analysis shows that the expected number of remote messages required to service a N-party page fault for the distributed pager is

- 3N-4 initially, and
- 2N-1 or
- 2N at steady state

depending on boundary conditions. To get the total number of messages in the distributed scheduler one must add a total of 2N-2 local messages between pagers and the kernels they service. For comparison, any centralized algorithm that maintains strict memory coherence must use at least 4N remote messages and no local messages. In the case of the simple scheduler this figure is 5N messages. Since the cost of local messages is often much less than the cost of remote messages, the distributed pager clearly outperforms the centralized one. The performance evaluation results, reported in Section 6 confirm this analysis.

4.1. Example

It is interesting to consider one example that shows the effects of forwarding page faults among distributed servers. Let us assume that N servers (each one serving one or more kernels) all take repeated page faults on the same page, which is the *hotspot* case that makes distributed shared memory perform the worst. Initially, all servers refer to the memory object's pages from the same one (say server 1). Therefore N-1 requests are sent to server 1. The server first services its local fault(s), then ships the page to server 2 (say) which becomes (in server's 1 opinion) the new owner. The next fault request is then forwarded by server 1 to server 2, the next to server 3 and so on, to server N-1. When all faults have been forwarded and served, the situation is such that servers 1, N-1 and N all know that the page is located at server N, while every other server *i* believes the page is at server *i+1*. When all servers take the next page fault only 2 requests are sent to the owner, and any other request *i* is queued at server *i+1* waiting for *i+1* itself to be served first.

```
S1      S2 ->   S3 ->   S4 ->   ...     Sn-1 -> Sn
|                                                ^
-------------------------------------------------
```

Figure 4-3: Steady State Behavior for a N-Party Write Hotspot

This situation is depicted in Figure 4-3 and can repeat itself. Our experiments show that indeed in a write-hotspot the system oscillates between two configurations of this type, never entering the initial state again. There is a worst case that could surface: an isolated page fault triggers a number of forwarding messages. This number is N-2, since always at least two servers know exactly where the page is: the owner and the one who sent the page to it. In the example, this would happen if server 2 alone takes a fault after the first N faults are served. After a worst case fault all servers know exactly where the page is, and therefore the system goes back to the initial state.

5. Related Work

Forwarding creates a new need: the need of forwarding page faults to the current owner of a page. Li [7] looked at the problem of locating a page and provided various algorithms to solve it, and analyzed their costs. Our distributed algorithm must be compared against the "Distributed Manager Algorithm 2.6", with the optimizations indicated at pages 61-63 that page invalidations are sent in a divide-and-conquer fashion. Note however that in Li's algorithms all operations are RPC, hence requiring twice as many messages and unnecessary serialization. Li also evaluates the use of broadcast messages and proves that they could benefit some of his algorithms, under the assumption that their cost is the same as a direct message. Note that in our algorithm the use of broadcasts would be detrimental to performance, since it brings back the system to the initial state and away from the most favorable situation. The idea of propagating invalidations in a divide-and-conquer fashion is, in our system, much more effective than broadcasts. In this paper it was only assumed that the underlying architecture provides efficient point-to-point communication, with quasi-uniform cost. The cost of sending a message to N recipients is therefore greater than or equal to N times the cost of a message to a single recipient.

Cheriton [3] has recently extended the V kernel to support user-level data and caching servers,

which can be used to provide distributed shared memory. His facility has many similarities with Mach's external pager facility, although it is described in terms of file abstractions rather than memory object abstractions. The implementation uses a scheme analogous to the simple scheduler presented above, but might add considerable extra message traffic by polling and forcing page flushes every T-milliseconds to provide *T-consistent* files for transaction support.

Fleisch [4] has extended the Locus kernel to provide distributed shared memory, with a SystemV interface. The scheme he describes seems geared to maintaining consistency at the segment rather than page level. A report on the implementation work will be necessary to better evaluate his approach. Other approaches to user-level shared memory are possible, [2] contains some references.

6. Performance Evaluation

The performance of the server was evaluated along a number of dimensions. Fundamental are the average times to service a fault, in both cases of single machine and multi-machine applications. These are affected by the various special features of the server. The centralized and distributed cases were compared, using programs that exercise the hotspot behavior. Our measures show two overall results: the distributed algorithm is more efficient than the centralized one, and none of the special features we introduced has an unacceptable impact on performance. The major bottleneck in the test configuration (token ring workstations) is the network latency, which accounts for about 98% of the elapsed times. More details on the measurements can be found in [5]. The server was instrumented in two ways: keeping track of the number and type of faults it services (per object and per page), and collecting extensive traces of the message activity. These data can be obtained via a remote procedure call by other processes, with minimum perturbation.

6.1. Basic Costs

The most common use of the server is in sharing memory within a single machine. In this case, a fault on a missing page (cache-fill) requires two local messages, for a total cost of 1.5ms on a IBM RT-APC. A protection fault also requires two messages but no memory mapping, for a cost of 1.1ms. A pageout operation requires two receive messages and the deallocation of data, which is not a system call but a RPC to the kernel and involves two messages. The total cost is then 2.5ms. Since system time is by far the dominant factor (93%) in all cases, schedulers do not show significant differences in the handling of local faults. Table 6-1 summarizes the most important costs.

6.2. Costs of The Algorithms

The multiple page size scheduler adds some overhead to the fault times, primarily because more server pages might be needed to cover a kernel's page fault. In most cases, a small range of page sizes will be used, but even with an unlikely ratio maximum/minimum page size of eight the overhead over the basic fault times is only 0.2ms. If necessary, however, the algorithm can be tuned further for larger page size ranges.

Various experiments were performed on the distributed scheduler, the most interesting one

Parameter	Measured Cost
Zero-fill Fault	1.5ms/fault
Protection Fault	1.1ms/fault
Hotspot Cycle	4.2ms/cycle
Multiple Page Size Overhead	0.2ms/fault max
Avg Messages, centralized hotspot case	5.0/fault (all remote)
Avg Messages, distributed hotspot case	4.1/fault (2.0 remote)
Forwarded Faults	10% (hotspot)
System Time	93%

Table 6-1: Costs Of The Server.

being the case of an hotspot page. This is demonstrated by a simple program that repeatedly increments the same memory location, replicated across various machines. The measures show that on average each server received requests for read/write/protection faults in an equal amount, as expected. The average number of messages per fault is the single most important figure: on average, each server handled 4.1 messages per fault. Half these messages are received and half sent. On average, 2.1 messages are local (interactions with the local kernel) and 2.0 are remote (interactions with other servers). This nicely confirms the estimates presented in Section 4. Remote messages are extremely more expensive than local ones: an average 98% overhead was observed in the test system, equally divided among the local Mach network server, the remote one, and the TCP/IP transfer protocol.

Machine	Int16/32	Float32	Float64
Sun 4/260 (*)	0.8	1.0	1.1
Vax 8800	1.5	2.3	3.7
IBM RT	1.9	2.4	2.5
Sun 3/280	1.9	2.5	2.9
μVax-III	2.8	4.6	6.8
Sun 3/160	3.0	4.8	4.6
Vax 785	4.4	7.6	10.9
Encore (*)	4.9	12.5	14.3
μVaxII	6.1	10.4	14.5
Vax 8200	9.1	15.3	27.9

Table 6-2: Overhead of Data Translations (in milliseconds per 4kbytes).

For the heterogeneity problem, only those machine types that are more or less implied by the definition of the C language were chosen for implementation, e.g. integers of various sorts and floating point numbers. Many other data types map obviously onto these types. For floating point numbers, the two formats that are most often used on our machines (Vax-D and IEEE-754) were selected. Both short (32 bits) and long (64 bits) forms were considered. Table 6-2 shows the

overheads measured on the server on a wide variety of machines. The times reported are those necessary to convert 4kbyte of data, but note that some machines use larger page sizes. There is no other basic overhead beyond a simple test of whether conversion is necessary or not. Starred entries in the table indicate machines for which a Mach External Pager kernel is not yet available. In these cases, a synthetic test was run to time the critical code.

Assuming that the server's (multi)processor has spare cycles, it is possible to eliminate the type conversion overhead at the expense of increased memory usage. The server keeps multiple copies of each segment, one per machine type, and pre-translates it when a page is received. Translation is done in parallel by a separate thread, which works in a pipelined fashion with the main thread that services faults. We have not yet implemented this optimization.

6.3. Application Programs

Intuitively, the performance gain from the use of memory sharing techniques comes from the large amounts of information that can be transferred with no cost between parallel activities in each synchronization operation. Below a certain threshold, on a uniprocessor the integration of scheduling and data transfer provided by a kernel optimized for message passing is apparent and wins over the simple busy-waiting scheme of spin-locks. The effect must be visible in the networked case, where spin-locks are more expensive. This was the idea that guided the choice of applications for testing the server. This hypothesis partially contradicts the suggestion that the *locality* of operations would completely dominate performance of a distributed shared memory program.

In the networked shared memory case, all the processes running on the same machine produce a single load on the pager, and the advantage of one process obtaining a page that will then be used by the other processes is not apparent. This non-measurable gain was eliminated from the experiments and only one process was allocated per machine even if this is clearly unfair to the pager.

Program	1 Machine	2 Machines	3 Machines
Matrix 128x128	29	15	10
Matrix 256x256	241	122	80
ShortestPath	60	60	40
LocusRoute	277	333	397
Mp3d	8.6	16.1	23.0

Table 6-3: Execution Times For Some Tightly Coupled Shared Memory Programs, Unmodified

All programs have been developed for a uniform shared memory multiprocessor, and were not modified in any way to get better distributed performance. In the matrix multiplication case, the problem is decomposed so that each machine computes all the elements of some row in the output matrix. In this way it is easy to compute large matrices with few processors. The Shortest Path program is a parallel version of a sequential algorithm which shows Nlog(N) complexity for

planar graphs [6]. The program evaluates in parallel the possible extensions to the most promising paths, and each activity only looks in the neighborhood of a point and queues the new extensions to other activities. The other two programs have been used in architectural simulations, on the assumption that they are representatives of a large class of parallel programs. Mp3d is a particle simulator [8] and LocusRoute is a parallel VLSI router [9].

The experiments were performed on machines under standard multi-user operating conditions, including a normal level of disk paging. Measures were taken of elapsed and per-thread CPU times. Table 6-3 shows the results of executing the programs on a small group of IBM RTs on a token ring. The network latency dominates performance, and only the matrix multiplication case shows linear speedup. All programs are known to demonstrate linear speedups on a bus-based shared memory multiprocessor with a small number of processors.

One important factor affecting the performance of an application that uses dynamically managed shared memory is the memory allocation algorithm used. Li described a scheme for memory allocation derived from Knuth's FirstFit scheme. A quick comparison was made with a different one, a descendant of Knuth's FreeList algorithm. Such an allocator is currently used, in a sequential version, by the standard Berkeley BSD Unix distribution. A parallel version was easily created by associating a semaphore to each free list, whereby requests for memory blocks of different sizes proceed completely in parallel. It is much more difficult to make the FirstFit scheme more parallel.

The measurements show that not only does the FreeList algorithm use less memory (1/4 on average) than the FirstFit one, but that it is about 20-30% faster even in the sequential case. Other measurements indicate that a two level memory allocation strategy is very effective in reducing shared memory contention. The simple solution of allocating and deallocating memory in batches for blocks of the most frequently used size often suffices to eliminate the most obvious bottlenecks.

7. Conclusions

This paper described a user-level memory server for Mach and the algorithms it uses for dealing with issues like heterogeneity, multiple page sizes, distributed service. The server itself shows very good performance, and the distributed algorithm is effective in reducing communication over the (potentially slow) communication medium. Results with application programs are dominated by the network latency, but still optimal in some cases. It is conjectured that the amount of data exchanged between synchronization points is the main indicator to consider when deciding between the use of distributed shared memory and message passing in a parallel application. There is definitely space for more research work: a number of extensions and optimizations can be attempted using more sophisticated caching strategies and heuristics in servicing fault requests.

Acknowledgements: We would like to thank Roberto Bisiani and David Black for their invaluable help in reviewing earlier drafts of this paper.

References

[1] Arnould, E., Bitz, F., Cooper, E., Kung, H.T., Sansom, R., Steenkiste, P.
The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers.
April, 1989.
To appear in the *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III).*

[2] Bisiani, R. and Forin, A.
Multilanguage Parallel Programming of Heterogeneous Machines.
IEEE Transactions on Computers , August, 1988.

[3] Cheriton, D.
Unified Management of Memory and File Caching Using the V Virtual Memory System.
Tech. Report STAN-CS-88-1192, Stanford University, Computer Science Department, 1988.

[4] Fleisch, B.
Distributed Shared Memory in a Loosely Coupled Distributed System.
In *Compcon Spring 1988.* IEEE, San Francisco, CA, February, 1988.

[5] Forin, A., Barrera, J., Young, M., Rashid, R.
Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach.
Tech. Report CMU-CS-88-165, Carnegie-Mellon University, Computer Science Department, August, 1988.

[6] Johnson, D.
Efficient Algorithms For Shortest Path Is Sparse Networks.
JACM 24(1):1-13, January, 1977.

[7] Li, K.
Shared Virtual Memory on Loosely Coupled Multiprocessors.
PhD thesis, Yale, September, 1986.

[8] McDonald, J.
A Direct Particle Simulation Method for Hypersonic Rarified Flow on a Shared Memory Multiprocessor.
Final Project Report CS411, Stanford University, Computer Science Department, March, 1988.

[9] Rose, J.
LocusRoute: A Parallel Global Router for Standard Cells.
In *Conf. on Design Automation*, pages 189-195. June, 1988.

[10] Sobek, S., Azam, M., Browne, J.
Architectural and Language Independent Parallel Programming: A Feasibility Demonstration.
In *International Conference on Parallel Programming.* IEEE, Chicago, August, 1988.

[11] Spector, A.
Distributed Transaction Processing and the Camelot System.
Distributed Operating Systems: Theory and Practice.
Springer-Verlag., 1987.

[12] Young, M., Tevenian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., Baron, R.
The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System.
In *11th Symposium on Operating Systems Principles.* ACM, November, 1987.

MINIMALIST PHYSICAL MEMORY CONTROL IN UNIX

Mark C. Holderbaugh
Scott E. Preece

*MOTOROLA Microcomputer Division**
1101 East University Avenue
Urbana, Illinois 61801

ABSTRACT

Computer systems often include areas of memory with special characteristics. Memory-mapped devices, memory boards with differing access times, and inter-system shared memory are typical examples. An operating system needs to provide (1) a mechanism for describing to the system the special characteristics of its memory, (2) internal mechanisms for using and managing the use of memory, and (3) external mechanisms allowing applications to use special kinds of memory.

The UNIX** operating system does not provide such mechanisms; it assumes memory is a vector of homogeneous storage locations. This paper describes extensions made to a UNIX system to provide the facilities described above through minimal changes to existing mechanisms inside the kernel and at the application interface.

The kernel maintains two new abstractions of memory objects: an *extent* is an area of physical memory with special properties defined by the hardware configuration; a *region* is an area within a specified extent, available to applications which need the special properties of the extent.

The system configuration mechanism has been extended to provide a vocabulary for describing memory extents and their special characteristics.

Regions may be defined either at system configuration time or dynamically and, depending on the definition of the containing extent, will be allocated either as a contiguous area of memory or as a group of individual pages. They are made available to applications as System V shared-memory objects which processes may map into their address spaces.

This paper includes a discussion of the needs of specific types of applications that deal with physical memory, a description of the user interfaces, and an overview of the implementation.

1 INTRODUCTION

Computer systems often include areas of memory with special characteristics such as speed differences, visibility on other processors, and device operation semantics. Memory-mapped devices, memory boards with differing access times, and inter-system shared memory are typical examples.

The UNIX operating system does not provide any mechanism for defining or using memory with special characteristics; it assumes memory is a linear, homogeneous array of contiguous storage locations. There is no notion of special semantics attached to particular areas of memory or of different costs associated with using different locations..

---

* At the time this work was performed the authors and the facility in which they work were part of the Computer Systems Division of Gould.

** UNIX is a trademark of AT&T.

Processes do not know the actual locations in physical memory associated with locations in their virtual address spaces.

To make effective use of special memory, the UNIX model must be extended to provide the following:

- A mechanism for describing to the system the special characteristics of its memory
- Internal mechanisms for using and managing the use of memory
- External mechanisms allowing applications to use special kinds of memory

Several UNIX vendors have approached this problem by providing facilities processes can use to change the mapping of parts of their address space [1] or to map designated storage objects into their address space [2, 3]. Such approaches require applications to have a low-level knowledge of the organization of their address space and to use proprietary mechanisms which are not portable.

An alternative approach is to support the use of special memory features through existing abstractions and interfaces on the process side. Physical configuration knowledge can be kept in a single place, the system configuration file, and its kernel-level representation. Applications can then be built to use physical memory resources through portable interfaces and can be debugged using conventional shared memory.

This paper presents the goals for the approach we took to supporting physical memory control, the key issues faced in implementing this approach, an overview of the implementation and user interfaces, and some comments based on our experience with the resulting system.

2 CONTEXT

The primary goal was to support real-time applications running on shared-memory multiprocessor systems. Applications would run on single or dual-processor super-minicomputer systems connected by inter-system shared memory. Pieces of an application might be written in C or FORTRAN; individual systems might be running either UTX/32*, Gould's version of the UNIX operating system, or MPX*, Gould's proprietary real-time operating system.

Customers were interested in moving to a UNIX environment because many program bids now require it, because they wanted increased portability, and because it provided a better development environment. At the same time, however, they had a strong interest in preserving existing code, especially in libraries and utilities, and in building applications in a style that was familiar to their designers.

Real-time MPX applications are typically built as a collection of FORTRAN programs using shared memory to maintain a common database describing the current state of the application as a whole. The database is built as one or more *Global commons*, a special kind of FORTRAN COMMON area that is mapped at run time into memory shared with other processes.

When the performance requirements of an application exceed the capacity of the two processors available on a single system bus, additional systems can be used, and specific areas of the application's data can be shared among the systems with *reflective memory*. Use of reflective memory requires the ability to map part of a process's address space into a specified contiguous area of physical memory.

---

* UTX/32 and MPX are trademarks of Gould, Inc.

When single-thread execution speed is critical and cache misses are unacceptable, high-speed non-cached memory may be required to make an application work. *Shadow memory* provides guaranteed cache-speed access but is very expensive and is usually reserved for critical program or data elements. Use of shadow memory requires the ability to allocate part of a process's address space in a specific area of physical memory, but contiguous allocation is not required.

3 GOALS

The primary requirement for the extensions was to support the use of reflective memory and shadow memory. The list of goals below includes the specific capabilities required for that support and some additional goals supporting enhanced usability and flexibility. These goals shaped the design and the specific capabilities we built.

- **Identification of special memory**
  It should be possible to specify to the kernel that particular areas of physical memory have particular characteristics. The system should not allocate special memory for general or kernel use.
- **Holes in memory**
  The system should recognize and use all memory that is present and not specified as having special characteristics. Addresses for which no memory is present should be skipped.
- **Use of physical memory**
  A process should be able to map areas of special memory into its virtual address space. The system should be able to perform a predefined set of such mappings for a process at **exec** time.
- **Separation of specification and use**
  The specification of hardware characteristics, which typically change infrequently, should be separated from the specification of the use of the memory, which changes as different applications run.
- **Multiple memory allocation models**
  The system should support alternative allocation models. Some uses of special memory require that specific physical locations be used for specific purposes; for example, a memory-mapped device would need to be mapped into a process as a contiguous chunk, so that transfers spanning memory management pages would map into the appropriate sequence of physical pages. Other applications only require that a particular type of memory be used; a process might want to have its text segment allocated in shadow memory, but would not be concerned about which specific physical pages were used.
- **Configuration independence**
  Applications should be insulated from hardware configuration changes, referring to memory objects by name rather than physical location. It should be possible to test an application on a single system using local shared memory, and then move the application to a multi-system environment using reflective memory without recompiling.
- **Portability**
  Applications should not have to use non-portable features to take advantage of special memory (unless, of course, they are doing non-portable things). Using memory control extensions to provide multiprocessor data sharing and to allocate high-speed memory to applications should not require building nonportable applications; such

applications should run, perhaps more slowly, on other standard UNIX systems.

- **Backward compatibility**
  All existing binaries should continue to work, except where their function specifically depends on system internals changed to support the extensions (that is, programs which don't look inside the system should not be affected by the changes).
- **Language support**
  All facilities should be available to either C or FORTRAN programs.

4 MEMORY EXTENTS AND REGIONS

To achieve our goals we built a new model of memory use, supporting much more flexible description and use of memory resources. Our model is based on two primitive notions of memory:

- *Memory extents* represent physical devices, such as memory boards and memory-mapped device interfaces, which occupy specific physical address ranges.
- *Memory regions* represent areas of the system's underlying physical memory which can be mapped into the address space of user processes.

Several properties of regions and extents are important to understanding their use:

- Each region or extent is homogeneous.
- Extents are static; they may only be defined at system configuration time (though they may have specific dynamic properties).
- Extents may not overlap.
- Regions may be dynamically created or deleted.
- Each region must lie entirely within one extent.

Our implementation provides:

- A method for describing regions and extents at system configuration time
- Tools and methods for creating and removing regions on a running system
- Tools and methods for attaching regions to user tasks
- Tools for seeing what regions are defined on a running system

5 DEFINING EXTENTS AND REGIONS

In UTX/32, a system configuration is built by the **config** program, which accepts a system description and produces files of data structure definitions. A kernel is built by compiling those files and linking them with the set of object modules required to support the described hardware configuration.

We extended the description language accepted by **config** to allow system administrators to specify which areas of physical memory to reserve from general use, the characteristics of such memory, how to handle the memory at startup time, and how to allocate the memory on request.

5.0.1 Memory extent description A memory extent directive defines the location and characteristics of an area of physical memory with the following syntax:

memory *name* **type** *type* **at** *addr* **length** *size*
[**control** *register_address*] [**key** *key*] [*attribute* ...]

memory *name*
Each extent must have a unique name so that other commands and specifications can identify it.

type *type*
The special semantics of a memory extent are determined by its class. Our implementation supports three types of memory: **shadow**, **reflective**, or **general** (used to reserve an area of normal memory for special use).

at *addr*
The address of the beginning of the memory extent. It must be an integral multiple of the page size.

length *size*
The size of the memory extent in bytes. It must be an integral multiple of the page size.

control *register_address*
Reflective memory extents have a memory-mapped control register associated with them; this value specifies the location of that register.

key*key*
Extents supporting noncontiguous allocation have a special kind of region associated with the whole extent (a *template region*, see below). *Key* is the region's System V shared-memory key.

attribute
Attributes give additional information about the extent. Some of them are mutually exclusive. *attribute* may be any of the following:

nocache
Caching is turned off for addresses in this memory extent.

noncontiguous
The extent supports noncontiguous allocation. Regions in the extent will consist of pages arbitrarily selected from the extent.

clear
The memory in this extent is to be cleared at system boot time.

probe
The memory in this extent is to be probed at system boot time. Any parity errors discovered will be corrected by zeroing the guilty word.

absent
There is no physical memory in the address range specified.

A memory extent unmodified by attributes is cached, contiguous, assumed to be present, and neither probed nor cleared at boot time.

5.0.2 Memory region description While memory regions can be dynamically specified, it is convenient to be able to define a default set of memory regions which are present at system startup. The system creates a System V shared-memory object with the appropriate name (and, optionally, numeric key). User processes may then attach to the object as usual.

A memory region directive has the following syntax:

region *name* **in** *memname* **at** *offset* **length** *size*
[**key** *key*] [**uid** *uid*] [**gid** *gid*] [**mode** *mode*] [*attribute* ...]

region *name*
Each region must have a unique name. Processes use this name to attach to the region.

in *memname*
The memory extent where the memory region is to be created. This extent must have already been declared via a memory extent directive.

at *offset*
The location of the region is specified as an offset from the beginning of the extent, so that extents may be moved without changing the definitions of the regions they contain. Since pages in **noncontiguous** extents are assigned arbitrarily, regions in such extents should always be at offset zero.

length *size*
The size in bytes of the region. It must be an integral multiple of the page size.

key *key*
Regions may be assigned a unique numeric, shared-memory key for use by System V portable code.

uid *uid*, **gid** *gid*
Access to regions is controlled by access modes; these are the UID and GID corresponding to owner and group in those modes.

mode *mode*
A number signifying, as for a file, the owner/group/other read/write/execute access permission modes for the region.

attribute
The attributes give additional information about the region. Currently only one attribute is defined:

initial-on
Indicates which of the regions of a reflective memory extent is to be active when the system is started.

6 USER INTERFACES

We considered several alternatives for the user interface structure, including building out the **mmap** interface sketched in the 4.3 BSD sources [3], or building a wholly new interface allowing user processes access to physical addresses [1, 2]. These approaches would allow tailoring the interface to the uses we had in mind, but would force applications to use proprietary interfaces. They also didn't support the idea of memory objects, which offered a lot of flexibility in reconfiguring memory from "outside" an application.

An alternative was to minimally extend the System V shared-memory interface so that shared-memory objects could be constructed to represent specific physical-memory areas. This approach would allow using existing, portable (if unappealing) interfaces and did support the idea of memory objects, but might not directly model the operations we had in mind.

We chose to build under the System V shared-memory semantics [4] because portability was an important goal, because we did not want processes to have to be aware of specific physical addresses, and because we did not want to mix the notions of devices or of memory-mapped files into the mechanism.*

Applications see memory regions as System V shared-memory objects. A process maps a region into its address space by doing a **shmget** [4] call to identify the object and a **shmat** [4] call to change the process's memory map to include the object's pages at the appropriate virtual addresses.

System V provides only integer keys as names for shared-memory objects. We felt this was obscure. We added names to shared-memory objects, and added a new system call, **shmgetbyname**, which is the same as the **shmget** call, except that a name is used, instead of a numeric key, to identify the region. Either form will work, assuming the region was created with both a name and a key. By allowing applications to use numeric keys to identify memory regions, we also meet our portability goal.

There is also a set of new system calls (Table 1) to create and destroy memory regions and to control reflective memory extents.

System Call	Parameters
mkregion	regname, memname, offset, size, key, uid, gid, mode, attributes
rmregion	regname
mem_reflect	memname, regname
shmgetbyname	name, size, shmflags

Table 1. New System Calls

* At the time this decision was made we also felt that the POSIX P1003.4 committee was likely to adopt the System V interfaces as the standard way of dealing with physical memory. That possibility has since become more problematic.

6.1 Creating and removing memory regions

The **mkregion** system call creates a memory region in a specified extent. The caller can specify the same information as described for the memory region directive defined for **config**.

The **rmregion** system call removes a memory region and all information associated with it from the system. Template regions, which are really handles for the pool of memory contained in an extent, cannot be removed.

6.2 Reflective memory support

Interprocessor memory sharing can be restricted to any region within a reflective memory extent. The **mem_reflect** system call specifies the region to be shared. Giving a NULL argument will turn off sharing of the extent.

6.3 Using memory regions

The System V shared-memory operations are sufficient to provide the required services to C programs, but they are not very convenient for setting up a complex environment.

We know of existing applications using hundreds of global commons; establishing them one by one would involve much code and many system calls. We wanted to allow the memory context to be defined at link time and to be established without explicit calls in the application, just as use of shared libraries should not require explicit user setup.

Further, while C applications can use the shared-memory operations described above to attach and detach from memory regions, FORTRAN programs cannot use the System V shared-memory interfaces because FORTRAN does not support pointer data types. FORTRAN programs can only use shared memory through global commons, which are supported as part of the language specification. Supporting global commons also requires the ability to specify the linkage to shared-memory in the executable, without explicit application code.

We considered doing this automatic setup in **crt0** or in a new preface section of the process process, without special kernel support. Doing the attachments at user level, however, would require system calls for each object attached, significantly slowing process startup.

The System V **exec** already included a similar feature for attaching shared libraries to a process's initial memory map. That implementation was not sufficient for our needs, since it didn't allow for initializing the objects from the executable, but it did provide a reasonable model for representing shared object names in the executable file.

We ended up building a generalization of the shared library facility, having the kernel handle the shared-memory object mappings as part of the **exec** operation, thus avoiding the extra system calls. The kernel obtains the list of mappings required from new sections in the COFF executable. The new entries specify the name of the region and the location in the user process's address space at which the region is to be attached. The design allows initial contents for the regions to be read from the executable, so that text, for instance, can be loaded into a shadow memory region.

The UTX/32 loader **ld** places global commons in separate pages in the program's address space. **ld** generates information in the COFF file indicating that at run time those pages are to be mapped into the named shared-memory object. If at run time there is no shared-memory object with the specified name, a regular shared-memory object is created with the name, allowing

programs to be debugged and tested in a local shared-memory environment.

6.4 Command level support

New user level commands provide wrappers around the **mkregion**, **rmregion**, and **mem_reflect** system calls (Table 2), so that users and system administrators can conveniently manipulate regions at shell level.

Commands	Options and Arguments
mkregion	[-k key] [-o offset] [-m mode] regname memname size owner group
rmregion	regname
mem_reflect	[-r regname] [-off] memname

Table 2. New User Commands

The System V **ipcs** command, which allows a user to view the current status of the shared-memory objects currently defined in the system, now can also print out the names of memory regions and shared-memory objects.

7 INTERNAL REPRESENTATION OF REGIONS AND EXTENTS

The most natural implementation of our new model would replace the core map (a linear array with an element for each physical page of memory) with a collection of data structures describing extents. A natural implementation would also replace the array of pages known to a given process with a collection of data structures describing regions known to the process. We considered a major revision of our memory management layer to directly support the System V notions of regions and pregions [4], which would provide convenient abstractions at the level we needed, but we decided that the changes would involve unacceptable risk of conflict with embedded assumptions in the rest of our kernel.

The files created by **config** define an array of memory extents and an array of shared-memory objects. At boot time, the kernel uses the array of extents to determine which areas of memory are probed and cleared and which are ignored.

The kernel builds a bitmap indicating the valid and invalid pages in the system. This bitmap provides a quick way for the system to determine if a page is present or absent.

The kernel then sets up the core map, which is still a linear array but now associates each element in the array with a particular extent. A separate freelist is built for each extent, including the pool of general memory. Each page's cmap indicates the extent in which it lies. For contiguous regions the freelist is maintained in order; for noncontiguous regions it is not.

Requests for pages are made for a particular extent; "normal" page faults are directed to the general memory extent.

The kernel also builds a paging map, which indicates which pages of memory are to be considered by the pager. Pages in memory extents or invalid/absent pages cannot be paged or swapped. The paging map allows the pager to avoid scanning the nonpageable pages.

The array of shared-memory object definitions defines the initial set of regions. It also contains a template region for each extent with the noncontiguous attribute. A template region has the same name as the extent it covers; requests to attach to a template region are filled by whatever physical pages are available in the extent.

The data structures describing shared-memory objects were expanded to allow them to describe regions. To preserve backward compatibility, however, the new information is not reported by the **shmctl** [4] IPC_STAT operation, which reports information on the size and ownership of shared-memory objects.

8 IMPLEMENTATION

The following list summarizes the low-level changes made to support the new functionality.

- **System startup**
  The existing memory management initialization process probed and cleared memory words until a nonpresent memory trap occurred. The startup routine was modified to deal separately with different areas of physical memory specified as having special characteristics, including whether the area is to be probed, cleared, or skipped. Startup gathers all nonspecial memory areas into a single pool for general use.
- **Memory management facilities**
  UTX/32 is built on a BSD memory management system, using **cmaps**. The existing code supported a single contiguous array of system memory pages. User processes were allocated pages from that map as needed. System V shared memory had already been built onto this system through a separate set of data structures; at attach/detach time the corresponding pages are copied into/out of the process's map. The memory management system was modified to fill memory page requests from the appropriate pool of pages.
- **Paging and swapping**
  The existing page daemon periodically looked at all of memory for stale pages to invalidate. The pager and swapper were modified to only page and swap general purpose, nonreserved memory; they ignore reserved memory and nonpresent memory. The pager was also modifed to request pages from the correct pool for the particular virtual address requested.
- **Process startup**
  The existing **exec** loaded the various parts of the executable file into specific areas of the process's address space (or set up fill-on-demand data so that the load happened as needed). **Exec** was modified to read the shared-memory descriptions out of the COFF file and include the corresponding memory pages in the initial memory map of the process.

9 EXPERIENCE

Our implementation has been used to port real-time applications to a UNIX environment and has met the needs of those ports. We were pleased to discover a number of non-real-time uses of the facilities as well.

The first use we made of memory extents was to support reservation of memory for a memory disk device. Once we had a way to keep the kernel from using an area of memory, it was easy to build a device driver to use that memory to emulate a disk drive. The memory disk device driver

has been bundled with the system; it provides disk-like semantics at memory-copy speed.

The memory-disk driver is also used as part of the distribution procedure: the distribution tape boots a kernel that loads a minimal filesystem into a memory disk, allowing the user to perform necessary configuration tasks on a running system without using disk drives (which may be at nonstandard addresses or may need to be formatted) at all.

The ability to control the amount of memory used by the system has proved useful in performance testing, allowing testers to run on systems of various sizes without physical reconfiguration. The results are only indicative, since they do not reflect the changes in interleaving that would be found on physically reconfigured systems, but are useful estimates.

The changes in the memory management code did not have any measurable effect on system performance.

Two important pieces of work remain to be done:

- The loader (**ld**) needs to be enhanced to handle a superset of the System V loader directives controlling placement of the pieces of the executable into the process's virtual memory space; it is hard to get the text of a process into shadow memory.
- The names of shared-memory objects currently constitute a separate namespace; this was a temporary expedient. There should be only one namespace and shared-memory objects should be named in it, along with files, devices, and various forthcoming classes of named objects.

10 SUMMARY

By implementing the user interfaces on top of System V shared memory we avoided designing a new set of data structures and operations; kept modifications of existing facilities to a minimum; and allowed users to build applications which would be portable but would be able to take advantage of certain kinds of performance boost when run on Gould's special memory products.

By extending **exec** to perform shared-memory operations from within the kernel we were able to extend special memory support to FORTRAN users, who would not be able to use address-based memory control.

11 ACKNOWLEDGEMENTS

The authors gratefully acknowledge the work of many others in the design and implementation of the memory classes facility. The implementation team included Caryn Cammarata, Paul Jones, and Denise Raffel. Denise also led the test team consisting of Morris Meyer, Dan Putnam, and Kendall Collett, who tested every aspect of the new facilities and of the modified memory management code. Gary Whisenhunt and John Gertwagen contributed to early designs and requirements analysis. Brian Marick repeatedly required the team to make sense and wrote the documentation with Vivian Smalley.

12 REFERENCES

[1] Edward A. Sznyter, Patrick Clancy, and James Crossland, "A New Virtual-Memory Implementation for Unix," *USENIX Conference Proceedings*, Summer 1986, p81-92.

[2] Robert A. Gingell, Joseph P. Moran, and William A. Shannon, "Virtual Memory Architecture in SunOS," *USENIX Conference Proceedings*, Summer 1987, p81-94.

[3] Marshall Kirk McKusick and Michael J. Karels, "A New Virtual Memory Implementation for Berkeley UNIX," *European Unix Users' Group Conference Proceedings*, Autumn 1986, p451-458.

[4] Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, 1986, p152-153, p367-370.

Software Configuration Management with an Object-Oriented Database

Eric Black
Atherton Technology
1333 Bordeaux Drive
Sunnyvale, CA 94089
ericb@atherton.com

ABSTRACT

Popular tools for software configuration management on UNIX™ and VMS™ such as SCCS, RCS, and CMS™ do not adequately provide for the management and control of large software configurations, nor for cooperative development among large numbers of programmers. Problems include awkward maintenance of multiple configurations, difficult enforcement of locally-imposed workflow rules, inability to coordinate and predict the impact of changes, and inconsistent follow-through on bug reports and change requests. These problems only increase with the introduction of more and more development tools such as structured analysis/structured design aids, documentation generators, publishing tools, requirements tracking tools, language compilers, analyzers and debuggers, each with its own input and output data which must be managed. An object-oriented database can be used profitably to solve these deficiencies.

This paper describes the configuration management features of Software BackPlane™, an Integrated Project Support Environment (IPSE). Examples of the problems which motivated the object-oriented configuration management model of Software BackPlane are given, showing how they are solved by that model, along with representative usage scenarios.

1. Introduction

Large software projects often encounter problems including errors in specification, programming and/or documentation, missed deadlines, and cost overruns, resulting in poor product quality and useability. These are caused in no small part by the special difficulties presented in simply managing configurations of such large amounts of data and large teams of people [BOEH81].

Current popular software configuration management tools used on UNIX[1] have limitations which prevent their being used over the course of the entire software development life cycle, and which present problems when dealing with large software configurations, moderate to large numbers of people, or both. These problems might be divided into three (overlapping) classes:

- **Tool Integration** (lack of uniform control over data)
- **Configuration Control** (difficulty in coordinating parallel and possibly conflicting development by cooperating groups or of multiple concurrent versions)
- **Work Flow Control** (difficulty enforcing local rules and methodologies for the development process)

---

[1] Trademark acknowledgment section: *Software BackPlane* is a trademark of Atherton Technology; *DEC/CMS*, *VAX*, and *VMS* are trademarks of Digital Equipment Corporation; *DSEE* is a trademark of Apollo Computer Corporation; *NSE* is a trademark of Sun Microsystems, Inc.; and, of course, *UNIX* is a registered trademark of AT&T.

A similar taxonomy is described in [SEAG81] as the functions of Source Management, Configuration Management, and Change Management, respectively.

In a mandated-methodology environment, such as encountered by developers who must follow DOD Standard 2167A [DOD88], the last problem area can be particularly important. Although it might be possible to solve each of these problems with appropriate standard sequences of user actions, methodologies which rely on correct user actions for each of many required steps are likely to be error-prone or even ignored. As a result, it can be difficult or impossible to guarantee reproducibility of a given configuration. On the other hand, if those user actions are embodied in programs or command scripts, such packages are likely to be quite specific to a particular project, and may require significant rework to be reused elsewhere, and represent a maintenance problem of their own.

Tools such as SCCS [ROCH75], RCS [TICH85], and CMS [DEC84] have been used successfully for configuration management of small to medium-sized projects, but they handle only ASCII text, and do not scale up gracefully to projects of a size which requires division into a large number of non-trivial parts. Because they deal mostly with individual files, they do not adequately address the problem of mutual compatibility of software components; that is, that a specific version of one file requires a specific version of another file. Although CMS has a "class" or "baseline" capability which facilitates grouping specific generations of different related elements in the same CMS library, and RCS has the ability to associate a "tag" with particular versions of a file and thereby extract the corresponding version for each of many files, significant additional effort is required on the part of the user when the project is of a size that does not conveniently fit into one CMS library or one RCS directory. Platform-specific collections of command scripts and programs such as SPMS [NICK] can help to organize projects which are so large that they must be broken into smaller pieces in order to be managed, but these systems do not necessarily address the problem of managing versions of these smaller pieces, and may not be readily carried over to another host platform.

If multiple configurations must be simultaneously accessible, as in the case of parallel development or continuing development with a product version in the field, it becomes much harder to make sure that all the right pieces are used in each case. The typical solution is to throw disk space at the problem [YOST85]. Tying together related or mutually-dependent versions of individual components is better addressed by configuration tools such as DSEE [LEBL85] and NSE [COUR88], [HOND88]. While these provide improved control over parallel development, they still are not easily extended by ordinary end-users to include new development tools and data, and tailoring for local conventions, rules, and development methodology is difficult or impossible.

Programming support environments such as Gandalf [HABE86] and CAIS [CAIS85], documentation support environments such as SODOS [HORO86] and generalized environments such as PCTE [PCTE86] attempt to address the need for extensibility and tailorability. But they do not provide a uniform model for configuration management and workflow control of software components encompassing all of requirements, specifications, source code, test specifications, test results, and documentation. And, they are still at the research level.

Recent work has been directed toward the development of an Integrated Project Support Environment (IPSE), encompassing support for all phases of the software product life cycle. A true IPSE provides far more than simply configuration management, although programmers perhaps see this as its major function. Other requirements which must be addressed [CATI88], [ELLI88] include:

- **Data Sharing** (between programs, tools, and project members)

- **Work Flow Control** (mechanisms for enforcing local policies and procedures)
- **Tool Integration** (accomodate varying software tools and their data, providing a whole which is greater than just the sum of the parts)
- **Portability** (of project members as well as programs and data)

Software BackPlane [PASE87, PASE88] is an IPSE which attempts to meet these requirements. It provides an object-oriented programming metaphor with an object type hierarchy, inheritance of object properties and method behavior, and object persistence in a distributed database. Client applications and tools can easily extend the built-in type hierarchy by creating new types which automatically inherit other properties and method behavior from the parent type, and providing only the required refinements. The built-in type hierarchy and methods include features necessary for the management of large projects involving potentially large numbers of cooperating project members, involved in many more tasks than just programming. The ease of extensibility allows users to integrate tools as needed, not merely as foreseen by the tool and/or environment vendor. It currently runs on UNIX and VAX/VMS platforms.

This paper focuses on the configuration management features of Software BackPlane. We give specific examples of the problems mentioned above, describe the approach taken in Software BackPlane toward their solution, and show typical usage of Software BackPlane in dealing with configuration management of large software projects.

2. Problems in Configuration Management

What are some specific examples of the kinds of problems that commonly crop up when managing large software configurations over the course of the development life-cycle?

Inconsistent version control: Because many tools provide their own storage and control for their data, we often see a different versioning mechanism for each tool's data. Some tools have no notion of version control at all; there is only one version of the data – the one that's out there – and you might get different, possibly nonrepeatable results depending on when you run the tool. Most UNIX-style filter (pipe-able) tools fall into this category. The lack of uniform control over data not only requires user re-training for each new mechanism encountered, but also often results in an inability to tie together the data for various tools, forcing the user to re-enter the same data in different ways, and resulting in numerous opportunities for error.

Lack of work flow control: It is hard to set up and enforce local conventions. Many UNIX software development sites are quite open; it is often possible for any user to check out, modify, and check in any file they want. This openness is fine in small groups, but it can become unmanageable in large projects. With SCCS and RCS, a user who can check out one file in a directory can check out any file in that directory. File and directory structure is usually set up along functional partitioning. While this partitioning is often isomorphic to areas of programmer responsibility and authorization, this is not always the case, particularly in large projects. Furthermore, local rules may permit the user to check out a given file at some times and not at others.

Typical work flow involves handing off logical (if not physical) control over version content to another group once certain criteria have been met (Figure 1). There must be control over when this handoff is allowed to occur, and changes must be prevented once the handoff has been made.

It is often required that source files follow particular rules as to format and content. For example, they might always contain a standard comment header. Neither SCCS, RCS, nor CMS have any notion of required contents of the text files they manage. Command scripts can be built to automate the inclusion of the standard header, but they do not enforce it.

Another example of work flow control rules is requiring C source files to pass lint with only allowable complaints. Others are comparing function interfaces to specification and documentation, requiring unit testing to have been run and acceptable results obtained, and signing off on bugs reported against a file being checked in. Such rules are difficult to automate using only standard facilities, and project members and managers are often reduced to checking them manually. This means they are not always followed, or followed correctly.

Difficulty managing parallel configurations: Parallel software development entails its own set of problems and difficulties. The term "parallel development" as used here encompasses both the case of multiple programmers working simultaneously on the same set of software components and the case of developing and maintaining multiple versions of a given set of software components, even by a single programmer.

Simultaneous parallel development often results in conflicting, mutually incompatible changes, or mutually required changes without explicit indication of the interdependence. There is often unexpected behavior, the result of changes made outside but which impact the work being done Think of the ubiquitous change to a global header file that "won't affect anything". Because it is difficult in most development environments to guarantee that no changes are imposed from without, it is difficult or impossible to guarantee reproducibility.

3. Model for Configuration Control

Our configuration management model is based on versioning, and on the distinction between mutable and immutable data. Versions are snapshots of data in time. Each version (except the initial version) has a predecessor, and may have one or more successors.

The concept of mutability as it applies to this model of versioning is crucial to configuration management. Versions are born mutable, that is, they may be modified as desired when created from their predecessor version (by "checking out" the predecessor). However, when the version is frozen (by "checking in" the new version), it becomes immutable, and its contents can never be changed. Instead, a new version must be created and modified as desired, up to the time when the new version is frozen (Figure 2). In this way it can be assured that any configuration which depends on a particular version cannot change unexpectedly.

The (checkout, checkin) pair may be thought of as delimiting a "long transaction". This is distinct from, but analogous to, the "short transaction" mechanism provided by the underlying database storage manager to regulate concurrent access to the database. Just as the short transaction prevents other database clients from seeing inconsistent data which is in the process of being updated, the long transaction protects other clients from unexpectedly seeing inconsistent data, or data which is still subject to change without notice.

There are occasions where it is convenient to refer collectively to a number of genealogically-related versions. The notion of a "line of descent" is already present in the linear sequence of predecessor version to successor version.

Parallel development often requires creating and maintaining parallel versions. We deal with this by "grafting" a new line of descent onto an existing one, sprouting from a specific version in the existing line of descent, thereby creating a *branch*. This means that the version may have more than one successor version. At most one successor version may be on the same line of descent.

In order to control visibility of change, new mutable versions are said to be in the *ghost* state. This somewhat whimsical term reflects the notion that while the contents of the version are mutable, they are not readily visible to all users, although it is possible to tell that the new version exists. It is possible for the contents of a mutable version to be visible to users other than

the creator of the new version, but in the normal case mutable data is not shareable. The checkin operation not only freezes a version and makes it immutable, it also makes the version sharable and its contents visible to other users.

Both SCCS edit files and locked checked-out RCS files are similar to ghost versions in that they start out mutable and can be checked in when the required changes have been made. However, both SCCS and RCS allow modification of a checked-in version under certain circumstances. Our model does not. Once a version is frozen by being checked in, it can never be changed. This implies that it can never be deleted. In actual practice this restriction is relaxed somewhat, so that frozen versions can be deleted if no other frozen version depends on them.

For many reasons it is desirable to partition a software project into distinct subparts (e.g., source file or file(s), documentation file(s), and test input and expected output). Each of these subparts may be individually versioned in order to track changes individually. It then becomes necessary to be able to refer collectively to the specific versions of the various subparts that make up a particular software configuration. Just as a UNIX file system defines a special kind of file called a directory, we define a special kind of version called a *collection*. Versions may be *attached* to a collection; an attached version may itself be a collection (Figure 3). The attachment is a reference only, and carries no implication of storage location. In many ways, a collection is analogous to a versioned directory containing symbolic links to the files (or directories) which are logically contained within that particular version of the directory. When the collection version is frozen by being checked in, it becomes immutable; that is, the set of specific versions which are attached to the collection cannot be changed.

If it is to be meaningful to make the contents of a collection version immutable by checking it in, the transitive closure of versions attached to the collection must also be immutable. In other words, in order to check in a collection, all attached versions must already be checked in. This means that checking in the collection at a top of a tree structure freezes the entire configuration in the subtree below that root. Because checked-in versions cannot be changed, any previous configuration can be re-created, as long as it was checked in.

Because ghost versions cannot be used by other users or configurations (except through explicit steps not detailed here), changes made by one user cannot impact other users. Those other users must, by definition, either be using ghost versions of their own or checked-in immutable versions (or both). This means, for example, that changes one user makes to a public header file do not cause unexpected problems (and recompilations) for other users. Once a user has frozen (and published) a version by checking it in, then other users can begin using that version. They can do this either by explicitly attaching the new version, or by attaching the "latest" version, meaning the last checked-in version on the same line of descent. In either case, the changes made by one user do not impact other users until they choose to allow it.

By definition, a copy of an immutable data item contains the same information as the original immutable data item. This makes it possible to cache local copies in a distributed database environment, and be assured that the cached copy is valid. Since the immutable data can never change, the local cache is never invalid. This notion of a "One True Source" is quite useful in software configuration management, and was nicely described in [LORD88].

The rules of this model for configuration management might be summarized as:

(1) Data is born mutable, but can be made immutable

(2) Once data is made immutable, it is always immutable

(3) Anything immutable is sharable
corollary: anything mutable is normally unshareable

(4) Anything immutable may not depend on anything mutable

(5) Change is inflicted only when asked for

(6) There is only One True Source

4. Object-Oriented Approach

We use an object-oriented approach to implement this configuration management model, and help address the above-mentioned problems and requirements encountered in large software projects. Among the advantages of such an approach are increased modularity and encapsulation of both data and code, enhanced reuseability of specifications and code via inheritance, and the potential for greater simplicity of design and implementation [STRO88]. These boil down to having to do less work when adding a new tool or when customizing for local work rules.

For example, the first implementation of Software BackPlane was based on an Entity-Relationship model, and made that E-R model visible at the external interface. That implementation required on the order of 450 different C functions to deal with configuration management, with the prospect of a combinatorial explosion in the interfaces with the addition of each new development tool.

Instead, we redesigned and re-implemented the publicly-visible interfaces following this object-oriented model. As a result, the new interface has fewer than 45 different functions, a tenfold reduction in complexity by that measure, although for convenience we provide a library with useful combinations packaged together. Due to the benefits of the type hierarchy and inheritance mechanism, adding a new tool to our development process and putting its data under Software BackPlane control requires no additional interfaces, and requires implementing only that code which implements new semantics of handling that data. Often zero new code is required.

Commonly-available object-oriented languages today include C++ [STRO87] and Smalltalk [GOLD83], but currently no implementions of these languages provide objects which persist beyond the life of the process or workspace session, although work is being done in that direction. A versioning model somewhat different from that described here has been implemented in AVANCE, a research object-oriented database [BJOR88]. There are few commercial object-oriented database systems available, including Vbase [ONTO86] and Gemstone [SERV87], but at this point because they are lacking in areas such as concurrency control and transaction management, they remain at the research or prototyping level.

Software BackPlane is designed and implemented using an object-oriented methodology, even though it is implemented in standard portable C.

4.1. Database vs. File

Why use a database at all?

After all, UNIX users have proven time and again that useful tools and combinations of tools can be constructed using only files to store data, and can even interchange data quite successfully. Tools designed and implemented so as to be useable in a pipeline are central to the toolset culture of UNIX [KERN76], [RITC87]. Easy redirection of standard input and output provided by the shell (which is perceived by many users as being part of the operating system) is essential, and applications taking advantage of the notion of stdin and stdout are much more readily combined. It is a significant advantage that the UNIX file system imposes no system-defined file formats and file types, that it is a "libertarian filesystem" [ODEL87]. However, filters are not the only useful form of tool, and the file-centered ASCII approach forces each

tool to solve several basic problems for itself.

Because there is no common data dictionary, there is only informal agreement as to data formats and their meaning. Most successfully combined tools in UNIX expect as input and send as output streams of ASCII text [ALLM87], [KERN84], and make no stricter assumptions about data format than that the data consists of a sequence of ASCII characters, with a New-Line at least once in every 255 characters. But for any kind of structured data, even if it is lines of ASCII text, inter-operability is only by gentleman's agreement. If the format of data output by the *ls* command changes, for example, to add new column of information, then all tools which accept as input lines of text from *ls* may need to be changed. The affected programs and/or scripts will not, in general, even be able to tell that the format of their input data is not as assumed. A database readily allows for centralized definition of datatypes and data formats in such a way that programs and tools can query for and handle whatever data format happens to be used (or at least know for sure when the data format is not one that is understood by the program).

A database allows for value-based or *associative* access and retrieval of data. For example, searching a file system for all C source files which contain a reference to a given external function can be quite expensive. A database allows such information to be stored explicitly as appropriate for the type, and the data dictionary allows applications to access this information in a uniform way. An alternative is to store such information in a separate file, such as is done in the TAGS file for GNU-emacs [STAL85], but because the format of this data is not described in a central location applications must "know" about it (gentleman's agreement), and the format cannot easily be changed without invalidating those existing applications. Information stored separately in this fashion is also quite likely to be out of date and incorrect.

File-based tools also are forced to deal individually with concurrency issues, such as file and/or record locking. Other programs which attempt to access the same files may in so doing completely bypass the concurrency control conventions and read invalid or inconsistent data, interfere with other tools by writing in spite of a supposed lock, or both. In addition, unexpected conditions such as a full file system or power failure will often cause data loss or corruption. Database access can provide a uniform transaction mechanism for dealing with concurrency control, including locking, atomicity of multiple updates, and fault tolerance.

4.2. Objects in Software BackPlane

This section gives a quick overview of the Software BackPlane notion of objects. For historical reasons, objects in Software BackPlane are called "elements"; in this paper, however, the term "object" will be used.

Objects have identity (instances are discernible from each other), have properties, and respond to messages. They also persist in a database which deals with details of storage allocation and management, and which provides transactions for concurrency control and atomicity of change operations. All objects understand and respond to three messages: **getProp**, which returns the value of a property [2], **setProp**, which sets the value of a property, and **free**, which instructs the object to delete itself and free any associated storage.

Every object is an instance of a type (also called a *class*). This idea is expressed in the "real world" when we say that a particular object "is a <fill in the blank>", for example, "this object

[2] This differs from the approach taken by some object-oriented languages, such as Smalltalk, in which a different message is used to fetch the value of each property, and indeed, existence of properties is essentially hidden behind messages.

is a computer" (is an instance of the class *COMPUTER*[3]).

In response to receiving a message a method function is invoked; this method function implements the semantics of the message as appropriate for the type. That is, the method function is associated with the class, and not with the instance. For example, there is a method function corresponding to the message "getProp" for the class *OBJECT*. All instances of that class, upon receiving the message "getProp", will invoke that same method function. The invocation of the method function is handled by a message dispatch mechanism. The logical connection between the (message, type) pair and the method function is made by a combination of data stored in the database and runtime binding of function addresses via a registration handshaking protocol.

This approach readily allows for encapsulation and hiding of the implementation, which for managing large projects is A Good Thing. It also fits in well with the tendency towards anthropomorphism that we computer types exhibit when speaking of programs, functions, and data. It is no longer a fanciful figure of speech to refer to "asking an item for its name", "telling an item to make a copy of itself", or "telling a version to check itself in".

With UNIX and other systems, it is common practice to define types by associating special meaning to particular file naming conventions, however loose the semantics of the type might be. For example, a file whose name ends in ".c" is often assumed to contain C source code. Software tools such as *make* [FELD79] take advantage of these conventions to select production rules. A language-sensitive editor might select a language template based on the name of the file being edited.

Such "metadata", or data about the data, can be erroneous or easily lost when based on assumptions such as file naming conventions. By representing the metadata explicitly in a database not only can the metadata be made more reliable, but applications can make use of it much more readily than is possible by being forced to search the file system for names and from them infer the metadata.

One special class, called *OBJECT_TYPE*, has instances which represent classes. That is, for every object type, there is a corresponding instance of the type *OBJECT_TYPE* which represents the type and describes what instances of that type look like. Instances of this special class understand the message *new*, which instructs the class to create a new instance. The method function which implements *new* for each particular type can do whatever type-specific initialization is required.

4.3. Type Hierarchy and Inheritance

Another important facet of the object-oriented approach is the notion of inheritance [STRO88]. Types may be refined by defining a *subtype*, and in so doing defining the ways in which instances of the subtype differ from instances of the parent type. In all other respects, instances of the subtype are like instances of the parent type; that is, the subtype *inherits* the properties, messages, and methods of the parent type. This idea is expressed in everyday speech in examples such as "DEC VAX is a kind of computer". This is not to be confused with the statement "This object is a DEC VAX, which is in turn a kind of computer." The latter statement shows that an instance of a subtype can also be considered to be an instance of the parent type.

A subtype can be refined in three ways: adding new properties, adding new messages (and corresponding method functions), and providing a new method function which implements messages which were inherited. The method dispatch mechanism used in Software BackPlane

[3] Classes or types will be denoted in this paper by *UPPER-CASE ITALICS*.

provides a simple way for the method function of the parent type to be invoked by the method function of the subtype, thus allowing re-use of code without copying code, or indeed, even knowing what code is being reused.

Applications can easily define new types by sending the *new* message to the *OBJECT_TYPE* instance which represents the type *OBJECT_TYPE* (that is, create a new instance of *OBJECT_TYPE*).

4.4. Contexts

In order to provide data shareability and still statisfy the requirement that change is inflicted only on request we introduce the notion of a *context*. All user work in the database is performed relative to a context. Contexts provide a single-user view of shared data. Contexts are analogous to a shell process or session which persists. They have a "root directory" (the top of a collection subtree), and a "current directory" (the current node in the collection subtree). Multiple contexts may refer to the same node. The context provides a mapping or projection of the directed acyclic graph of versions attached to collections into a tree structure. Although a particular node may be attached to multiple collections, at most one of those collections may appear in the collection subtree as viewed from the root (Figure 4).

Ghost versions are associated with the context relative to which the *reserve* message was sent to the predecessor version. Ghost versions are not allowed to be attached to any collection subtree other than the one pointed to by the context with which they are associated (colloquially, versions are "checked out in a context", and may not appear in any other context).

A new kind of object property (or "instance variable") is introduced, called the *context-sensitive property*. Context-sensitive properties may return different values when queried from different contexts.

An example of a context-sensitive property is the ".o" file resulting from compiling a particular C source file. This ".o" file is obviously associated with the C source; it is reasonable to consider the bits contained in the ".o" file generated by the compiler as being a property of the C_SOURCE object which contains the ASCII text of the C source code which was fed to the compiler. Even if the C_SOURCE object is "checked in" and immutable, various external influences can cause the resulting ".o" file to be different in different invocations of the compiler. For example, in one context the C file may be compiled with the command-line define "**-DEOL='\n'**", and in another with "**-DEOL='\r'**". We can associate with the context the particular set of command line flags to be used in invoking the compiler (by making that a property of the context), but we can have two different contexts referencing the same checked-in and immutable C_SOURCE object. We can store the ".o" output of the compiler as a *context-sensitive* property of the C_SOURCE object, and obtain the correct value in each case.

Another example of a context-sensitive property is the "parent" property on a *VERSION* instance; this is the inverse of the multi-valued "children" property on *COLLECTION*, which has as its value the object handles for all versions attached to the collection. Because the context projects a directed graph of collection attachments onto a tree structure, this property may return a different value depending through which *CONTEXT* the question is being asked.

As another example of a use of a context-sensitive property, consider that most tools are unaware of the database, and require input and output to be in files. In order to accomodate these tools, we can create a file system directory and store its path in a context-sensitive property of the collection. Then, by refining the *attach*, *detach*, *reserve*, *replace*, and *unreserve* methods to automatically export the contents of the attached version to a file in the *collection directory*, we have files which can be accessed by such tools, with the contents of the files under our control. If the collection directory path property of the collection were not context-

sensitive, then two users sharing the same checked-in collection (such as a subsystem library) would collide.

5. Usage Scenarios

So now, how can all this be used for software configuration management? This section describes some typical user scenarios, and shows how the requirements for configuration management are served by the object-oriented design.

Single user, single line of development: This is the simplest case. First the user creates a work context; this is analogous to logging in and starting a shell. The user then creates a collection which is to contain the various components to be worked on, and makes that collection the top of the collection tree referenced by the context (by setting the "top" property of the context object with a *setProp* message); this is similar to creating a directory to work in. The user creates a ghost version of the collection by checking it out with the *reserve* message; the "working directory" is now modifiable. The user creates and attaches components and/or subcollections as necessary. The *new* methods for the types of the new objects may be refined so as to automatically reserve the empty initial version and import a standard template, and perhaps to automatically create and attach additional standard versions, such as test drivers and approval lists. The user checks out each attached version to modify it, and checks it in to freeze its current state.

When ready to release (or pass it on to the next step in the development work flow), the user freezes the entire configuration by recursively replacing all ghost versions all the way up to the top collection (a "nested" checkin). Again, the *replace* method may be refined so as to verify that local rules are followed, such as requiring a C module to pass "lint" checking without complaints (with allowable complaints being filtered from the lint output), or confirming that the corresponding test case has been run. The *reserve* method for the collection may also automatically notify the next work group that the configuration is ready for integration and testing.

Once checked in and made immutable, all versions in the collection tree may be referenced by other contexts. For example, a separate context may be set pointing to the collection for the purposes of testing the release. Any compilations and/or testing in that context of the versions in the frozen configuration will not affect the view of the frozen configuration in the development context.

The user continues development by checking out the top collection, creating a new ghost version, and leaving the checked-in version used by other developers or testers unchanged. If local rules prohibit continuing development until the integration test and release procedures have run their course, the *reserve* can be refined to check to see whether checking out is to be allowed.

Multiple user, single line of development: The previous example is easily extended to the case of cooperating group members working on separate subparts in a single line of development. For example, if the top collection for one development context represents a subsystem, the same collection version may be attached to another collection in an integration context.

Single user, multiple line of development: For example, continuing development while maintaining product in the field. The user in the above example has checked out the development collection and started making changes. In the meantime, a bug has been reported in the field and must be fixed before the next full release. No other changes the user may have made are permitted to go out with this bugfix release.

The user creates a new context for fixing the bug, and points its "top" at the collection version for the product in the field. Because this collection already has a successor version on the same line of descent, it can only be checked out by creating a branch, perhaps using the name of the bug report or SPR as the branch name. Refinement of the *reserve* method for the collection can enforce such conventions. The new ghost version of the collection starts out with the same attachments. The user checks out the attached version or versions in order to fix the bug; again, local convention may require making these changes always to branched versions.

The user checks in the collection tree, which because of method refinement may require prior re-verification of test results and electronic approval signatures. The changes which fixed the bug will probably need to be made in the development versions as well. Lacking a facility for "parallel variants", where such changes might be applied automatically if there are not conflicting changes, we can refine the *replace* method for the component to check the predecessor versions back to the initial version to see if any new branches have been created, notify the user if any are found, and refuse the checkin (a property is defined on the component version to store the time the user last acknowledged new branches).

Multiple user, multiple line of development: This can often arise with cooperating group members working on overlapping subparts. The preceding examples have shown the necessary pieces to handle this case. Each user works in his own context. All may start out with their context "top" pointing to the same collection version. It may help for the initialization of the collection contents from templates to be done before the branching; this would have to be done, and the collection checked in, before other users attempt to reference the collection from their contexts.

Each user checks out the collection creating a branch, keeping the main line of descent of the collection available for merging the various pieces before release. Again, refinement of the *reserve* method on the collection can enforce this convention. Each user checks out and modifies the attached versions which are his responsibility.

The group periodically merges its work together into the collection version on the mainline which has been reserved for that purpose (in yet another context). Each group member first checks in any attached versions that may be checked out. The "diff" operation on two collections shows how the two differ. The "merge" operation on the collection uses the difference information and user input to update attachments in the target collection, so that the merged collection contains the desired versions.

If multiple users must make changes to the same component (e.g. a shared header file), each must check it out on a branch, make the necessary changes, and check it in. The merge operation now consists of more than just the collection merge; if the collection "diff" shows that the two collections contain a component with versions on two different lines of descent, then a "content merge" must be done on that component, merging the sum of the changes into one version which is in turn attached to the merge collection

After each merge of the group's work, the group members will probably wish to perform the same collection merge operation with their own development collections, instructing the merge to always select the version attached to the (mainline) merged collection if the attachments differ.

6. Experience

At Atherton we have been using Software BackPlane for in-house development and to maintain itself since April 1988. We have a network with 16 VAX systems running either Ultrix or VMS, and 27 Sun-3 workstations running various Sun-OS versions of UNIX, with 19 developers and 12 QA/test personnel. Total lines of source code for the product version now in the

field is approximately 658,000, with total cumulative (unique) lines of code for the current and previous versions (including unreleased development-only versions) of approximately 2,568,000 lines. We have 9 databases divided among various developer groups, and one large database containing integrated versions of subsystems. The number of collection versions in the development databases varies from 20 to 150, and the total number of non-collection versions ranges from approximately 200 to 4600. In the integration database there are approximately 270 collection versions, and approximately 3800 non-collection versions. Our in-house databases have 51 different object types defined, or 22 application-defined types over and above the 29 built-in types (such as *OBJECT*, *OBJECT_TYPE*, and *VERSION*).

Some of our groups must deal with continuing development involving potentially conflicting changes to shared files and maintenance of versions in the field resulting in 12 or more branch lines of development just for one subsystem. Tracking changes given such requirements is a challenge with the tools usually available for UNIX or VMS development. Disk space requirements for keeping all available online in normal directories and files would be rather large, and the probability of error in managing this development effort would otherwise be unacceptably high. The bottom line for these groups is that without Software BackPlane, the configuration problem would be a nightmare.

7. Future Work

Plans for future work on the Configuration Management aspect of Software BackPlane include:

- Re-implement using an object-oriented language, such as C++
- An interpreted language for refined methods and ad-hoc queries, perhaps based on SQL
- Heterogeneous access to databases distributed across multiple host platforms
- Variant branches -- automatic application of changes in parallel branch

8. Conclusion

An object-oriented approach can prove very helpful in managing large software configurations, especially where new tools and new types of data are continually being added to the development process. It is impossible to foresee all desirable tools or combinations of tools, so a mechanism for accomodating new ones as they come up is a requirement.

The combination of the object-oriented metaphor with the persistence and concurrency control of a commercial-quality database provides a powerful environment for configuration management. In actual use it has shown benefits that are difficult or impossible to obtain using commonly-available tools.

In addition, the benefits of the object-oriented approach can be obtained without the use of unusual or new programming languages or operating systems. Such a configuration management environment is available in Software BackPlane, which is programmed in C, and runs on both UNIX and VMS.

9. References

[ALLM87] Allman, E., "UNIX: The Data Forms", in *Proceedings of the Winter 1987 USENIX Technical Conference*, Washington, D. C. (January 1987)

[BJOR88] Bjornerstedt, A., "Version Control in an Object-Oriented Architecture", SYSLAB Report No. 57, University of Stockholm (May 1988)

[BOEH81] Boehm, B., *Software Engineering Economics*, Prentice-Hall (1981)

[BOOC86] Booch, G., "Object-Oriented Development", *IEEE Transactions on Software Engineering*, SE-12(2) (February 1986)

[CAIS85] Ada Joint Program Office, *Military Standard Common APSE Interface Set (CAIS)*, AD-A157 589, National Technical Information Service (January 1985)

[CATI88] Atherton Technology, Digital Equipment Corporation, et. al., *Common Application and Tool Integration Services*, work in progress, to be published

[COUR88] Courington, W., Feiber, J., and Honda, M., "NSE Highlights", *Sun Technology: The Journal for Sun Users*, (Winter 1988)

[DEC84] Digital Equipment Corporation, *User's Introduction to VAX DEC/CMS*, Order No. AA-L371B-TE (1984)

[DOD88] Department of Defense, *Defense System Software Development*, Military Standard DOD-STD-2167A (Feb 1988)

[ELLI88] Ellison, R., "Trends in Software Development Environments for Large Software Systems", in *Digest of Papers from the Spring 1988 COMPCON*, San Francisco March 1988.

[FELD79] Feldman, S., "Make – A Program for Maintaining Computer Programs", *Software Practice and Experience* (April 1979)

[GOLD83] Goldberg, A. et. al., *Smalltalk-80, The Language and its Implementation*, Addison-Wesley (1983)

[HABE86] Haberman, A. N., and Notkin, D., "Gandalf: Software Development Environments", *IEEE Transactions on Software Engineering*, SE-12(12) (December 1986)

[HOND88] Honda, M., Feiber, J., and Courington, W., "Type Extensibility in the Sun Network Software Environment", in *Proceedings of the Systems Design and Networks Conference*, Santa Clara (April 1988)

[HORO86] Horowitz, E., and Williamson, R. C., "SODOS: A Software Documentation Support Environment – Its Use", *IEEE Transactions on Software Engineering*, SE-12(11) (November 1986)

[KATZ86] Katz, R., Chang, E., and Bhateja, R., "Version Modeling Concepts for Computer-Aided Design Databases," ACM SIGMOD Proceedings, Washington, DC (June 1986)

[KERN76] Kernighan, B. W., and Plauger, P. J., *Software Tools*, Addison-Wesley (1976)

[KERN84] Kernighan, B. W., "The UNIX System and Software Reusability", *IEEE Transactions on Software Engineering*, SE-10(5) (September 1984)

[LEBL85] Leblang, D. et. al., "The DOMAIN Software Engineering Environment for Large Scale Software Development Efforts," in *Proceedings of the IEEE Conference on Workstations*, San Jose (November 1985)

[LORD88] Lord, Thomas, "Tools and Policies for the Hierarchical Management of Source Code Development," in *Proceedings of the Summer 1988 USENIX Conference*, San Francisco (June 1988)

[ONTO86] Ontologic Incorporated, *Vbase Functional Specification for Release 0.8* (1986)

[NICK] Nicklin, P., "The SPMS Software Project Management System," documentation included in source for User-Contributed Software, 4.2bsd, University of California at

Berkeley

[ODEL87] O'Dell, M., "UNIX: The World View", in *Proceedings of the Winter 1987 USENIX Technical Conference*, Washington, D. C. (January 1987)

[PASE88] Paseman, W. G., "Architecture of an Integration and Portability Platform," in *Digest of Papers from the Spring 1988 COMPCON*, San Francisco March 1988.

[PASE88] Paseman, W. G., "Requirements and Architecture for Tool Integration," in *Proceedings of the Systems Design and Networks Conference*, Santa Clara (April 1988)

[PCTE86] Esprit, *PCTE, A Basis for a Portable Common Tool Environment – Functional Specifications*, Fourth Edition, Commission of the European Communities, Brussels (1986)

[RITC87] Ritchie, D., "Unix: A Dialectic", in *Proceedings of the Winter 1987 USENIX Technical Conference*, Washington, D. C. (January 1987)

[ROCH75] Rochkind, M., "The Source Code Control System," *IEEE Transactions on Software Engineering*, SE-1(4), (December 1975)

[SEAG81] Seagraves, D., and Sagan, J., "Configuration Management in Large Software Products", in *Proceedings of the 10th International Switching Symposium*, Montreal (September 1981)

[SERV87] Servio Logic Development Corporation, *Programming in OPAL*, (1987)

[STAL85] Stallman, R., *GNU Emacs User Manual*, Fourth Edition, Emacs Version 17, Free Software Foundation (February 1986)

[STRO87] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley (1987)

[STRO88] Stroustrup, B., "What is Object-Oriented Programming," *IEEE Software*, IEEE (May 1988)

[TICH85] Tichy, W. F., "RCS – A System for Version Control," *Software Practice and Experience*, 15(7) (1985)

[YOST85] Yost, D., "The Cloned Tree Method of Revision Control, or A Rich Person's Revision Control System", in *Proceedings of the Summer 1985 USENIX Conference*, Portland (June 1985)

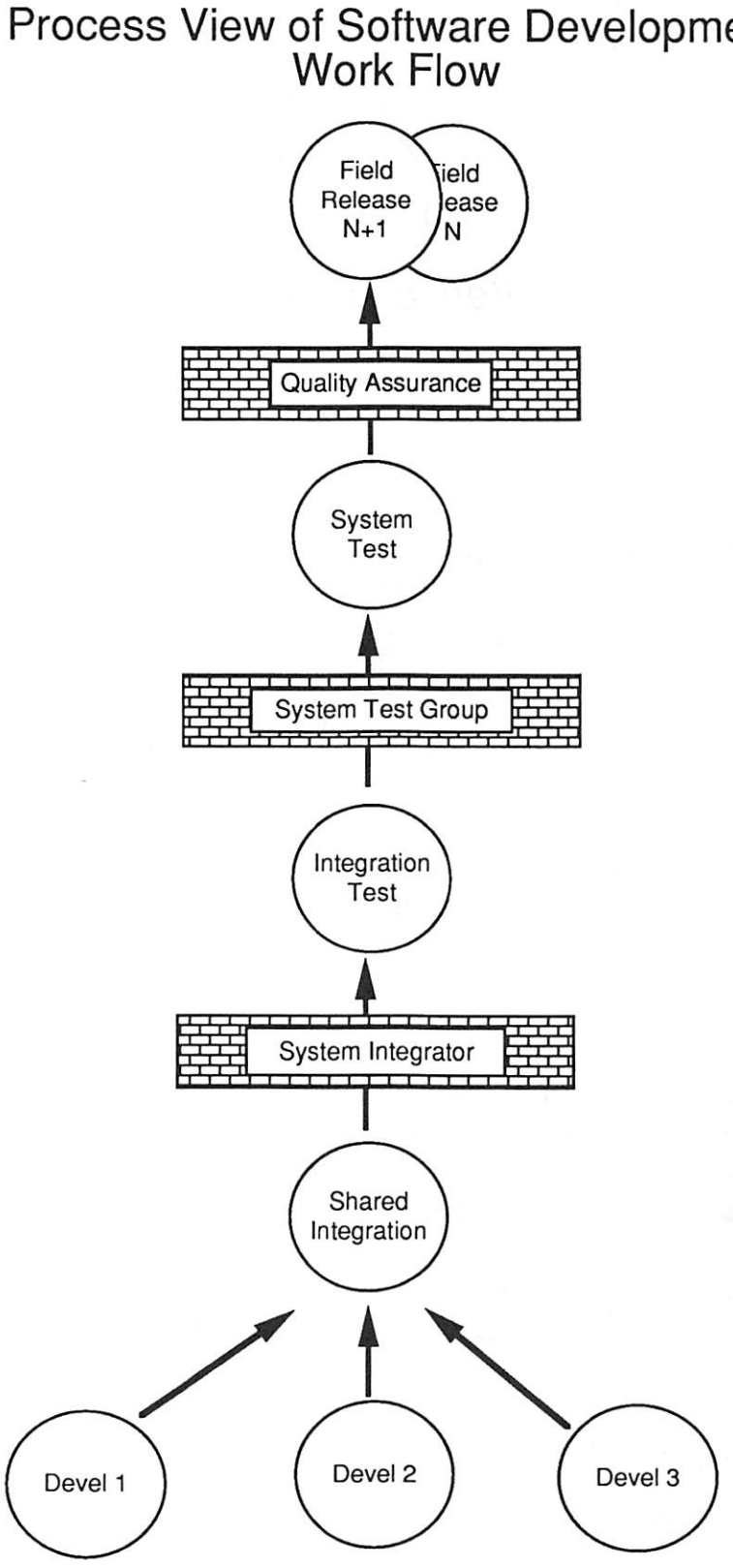
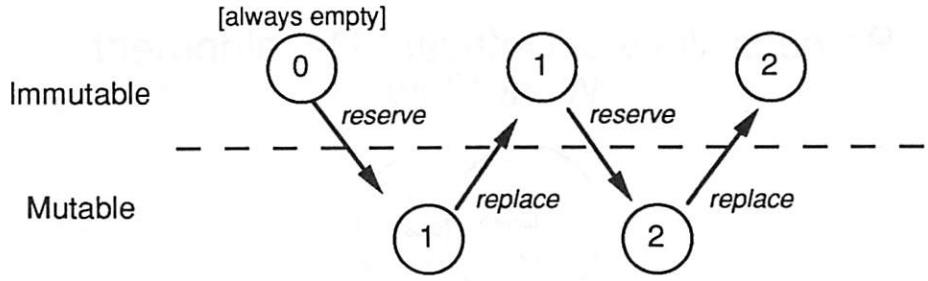
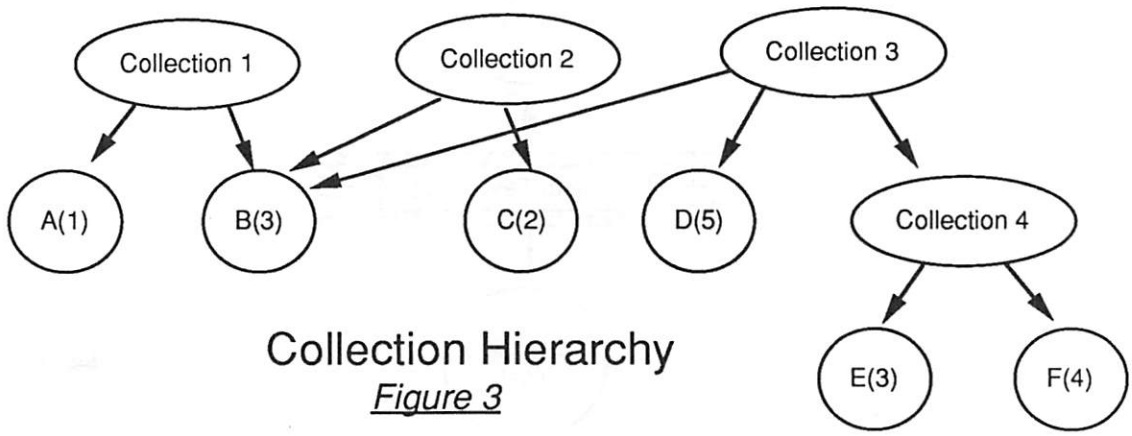

Process View of Software Development
Work Flow
Field
Release
N+1
Field
Release
N
Quality Assurance
System
Test
System Test Group
Integration
Test
System Integrator
Shared
Integration
Devel 1
Devel 2
Devel 3

Figure 1



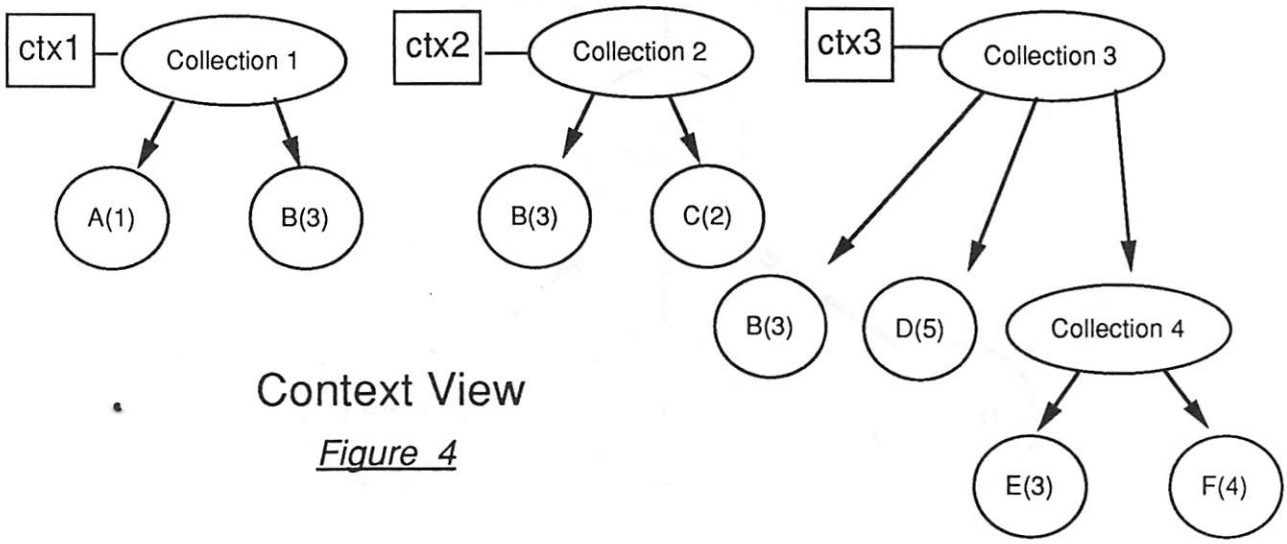
Versioning Model

Figure 2



Collection Hierarchy

Figure 3



Context View

Figure 4

Supporting Objects in a Conventional Operating System[1]

Prasun Dewan (pd@purdue.edu)
Eric Vasilik (vasilik@purdue.edu)

Department of Computer Science
Purdue University
W. Lafayette, IN 47907

ABSTRACT

A simple approach is presented for introducing objects in a conventional operating system. Objects are created as combinations of conventional processes and files. Like processes, they are active agents capable of executing code and communicating with other objects. Like files, they are persistent, have a protected name in a file system, and are opened and closed for access. Motivation for supporting objects in a conventional operating system is presented. The basic elements of our approach are described together with the rationale for our decisions. An implementation on Unix[2] is discussed. Object-based programming at the system level is contrasted with object-based programming at the language level and comparisons are made between our approach and related work.

1. Introduction

In a conventional operating system, files and the processes that manipulate them are kept separate. This separation leads to at least three problems:

- Any process that has access to a file may manipulate it. As an example, consider an appointment file that stores a list of appointments for a user. Only processes executing programs written specifically to manipulate such a file should be allowed to access it. However such a file may also be accessed by compilers, text formatters, and other processes having appropriate access rights.
- Each process that correctly manipulates a file needs to know the syntactic and semantic constraints of the data stored in the file. This information is hard coded in the program executed by the processes. As a result, it is hard to change the syntax and semantics of data stored in files. Continuing with the appointments example, each program that manipulates an appointment file needs to detect syntax errors such as a missing appointment time, and semantic errors such as an appointment time that has elapsed. If the semantics of appointment lists were changed to, say, require that a change to an appointment in one file is reflected in the appointment files of all other users involved in the meeting, then all

[1]This research is supported in part by the NSF sponsored Software Engineering Research Center.

[2]Unix is a registered trademark of AT&T Bell Laboratories.

programs that manipulate appointment files need to be located and changed to incorporate the change.

- At any time, a copy of the information in a file may also be loaded in the data structures of a process. These copies may be manipulated independently, and can, therefore, become inconsistent. Moreover, attempts to keep these copies consistent may lead to wastage of computer resources, since the process needs to "poll" the file for changes made by other processes. For instance, a process that displays a user's appointements for the current day must read the appointments file periodically to ensure that the information it displays is consistent with the current contents of the appointments file. Reading the file frequently would consume computer resources while reading it infrequently can cause a user to miss an appointment.

Operating systems such as Demos [2], Charlotte [8], Unix, and Mach [9] that support the client-server paradigm of interaction have illustrated the use of object-based programming to solve some of these problems. Information that would otherwise be kept in a file may instead be encapsulated in the data structures of a *server* process, which responds to messages from clients wishing to manipulate the information. The server is solely responsible for managing the information and defining its syntax and semantics.

Returning to the appointments example, an "appointment server" may be created for each list of appointments. Processes wishing to manipulate such a list send messages to the server, which makes the requested changes according to the syntax and semantics of the list. An example of such a message may be the **add_appointment** message, which a client may send to add an appointment to the list. A server may process this message by checking the new entry for errors, such as a missing or invalid appointment time, adding the entry in case of no errors, and sending messages to appointment servers of other users involved in the meeting. The definition of the syntax and semantics of appointment lists is coded in the program that the appointment servers execute, and may be changed by modifying only this program.

The client-server model of process interaction, however, is not by itself sufficient for encapsulating all information:

- Servers are temporary process and thus unsuitable to encapsulate persistent information. A persistent server can be simulated by temporary *incarnations* of the server. Each incarnation is a new process that executes the server's program and checkpoints its persistent state in a data file, which the next incarnation can read to initialize its state. A problem with this scheme is that any process with appropriate access rights can modify the data file. Moreover, errors can be made in connecting an incarnation to its data file.
- Access to servers is not protected from arbitrary clients. Most server-based systems do require that a client have a capability to a port advertised by a server before it can send a message to the port. However, typically, the operating system distributes these capabilities to clients without distinguishing between different classes of clients. For instance, Berkeley Unix allows any process to connect to a socket on which a server is listening. Similarly, a Demos or Charlotte name server gives a link registered by a server to any client that requests it. As a result, secure information in these systems needs to be accessed via protected files.
- An incarnation of a server needs to be always active since requests from clients may arrive at random times. However, only a limited number of processes can be active at any one time. Therefore, only a small amount of information can be accessed through servers. The majority of information needs to be accessed via files.

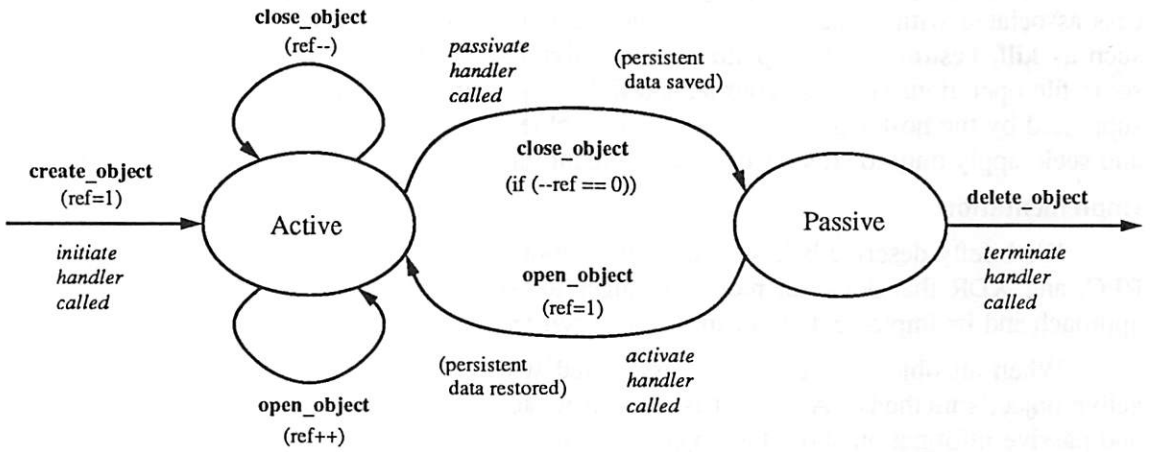


Figure 2
Object State Transition Diagram

- The machine on which the client that activates it resides. This approach guarantees that at least one process communicating with the object is on the same machine as the object. However, it requires that the object's class be executable on each machine on which it is opened. Moreover, it does not accommodate programs using machine-specific file names since they may be executed on more than one machine.
- The machine on which the object name resides. Thus object "/usr/joe/appts" would always execute on the machine on which the directory "/usr/joe" resides. It would be the owner's responsibility to ensure that the object's class is executable on this machine. This approach is consistent with file activation schemes supported by distributed file systems such as NFS which activate files on machine on which their names reside. Moreover, it lets a user migrate objects by renaming them. (Migration of active objects may be disallowed if the underlying system does not support process migration). However, it is possible for an object to be activated on a machine on which none of its clients reside. This situation would occur frequently when objects are created and opened from diskless workstations. Moreover, since objects may migrate, this approach also does not allow the use of machine-specific names.
- The least loaded machine on which the program can execute. This scheme requires the complexity of determining which machine is least loaded. Moreover, it is not useful when compute/communication ratio of the object is small, and does not allow the use of machine-specific names.
- The machine on which the creator of the object is executed. We have chosen this approach in our current implementation because it accommodates programs using machine-specific names. A potential drawback of this approach is that none of the clients communicating with the object may reside on the creator's machine. However, this situation would occur frequently only for those objects that are frequently accessed by several hosts. We expect the majority of objects, like files, to be used mainly by clients

executing on the creator's machine.

Other Operations on Objects

A client may invoke the **pid_of_object** operation on an object to get the *pid* of the process associated with an active object. The *pid* may later be used to invoke process operations such as **kill, resume,** and **suspend** on the active object. Since objects are in the file system, some file operations such as **rename,** and **link** may be invoked on them. Not all file operations supported by the host system are applicable to objects, since some of them such as **read, write,** and **seek,** apply only to streams of data (a stream interpretation of objects is discussed in § 5).

Implementation

We briefly describe below an implementation of our approach on Unix running Sun NFS, RPC, and XDR that does not require changes to the kernel. In section 3 we discuss how the approach and its implementation can be enhanced by making changes to the kernel.

When an object is active it is associated with an an *object process* which executes an active object's methods. An object is also associated with an *object file* which keeps the active and passive information about the object. The former consists of the identifier of its object process, its reference count, and a Sun RPC program number used for sending messages to it. The latter consists of the object's class name, name of its data file, and home (host on which it was created).

Each host runs an *object manager* which creates, activates, and passivates objects on that host. The operations on objects are implemented as library routines linked to client programs which communicate with object managers.

An object file is created by the **create_object** operation, accessed by **open_object**, **close_object**, and **pid_of_object** operations, and deleted by the **delete_object** operation. It is associated with the permissions specified for the object in the **create_object** call. As a result, the following protection scheme is defined for objects: A client can create and delete objects in a directory only if it can create and delete files in that directory. A client can invoke the **pid_of_object** operation on an object only if it has **read** access to the object, since this operation needs to read the object file to return the pid of the object process. It can invoke the **close_object** operation on an object only if it has **write** access to the object, since this operation needs to modify the reference count of the object. Similarly, it can invoke the **open_object** operation on an object only if it has both **read** and **write** access to the object.

The name of an object file the same as the name of the corresponding object. As a result our implementation uses the underlying implementation of a hierarchical network file system to resolve object names. For instance, the **open_object** operation simply opens the corresponding object file to determine the object to be opened. Moreover, the implementation of the file operations **link,** and **rename** is directly used for objects. For instance, the operation

link ("/usr/joe/appts", "/usr/jack/joe_appts")

which creates the new alias ''/usr/jack/joe_appts'' for the object file ''/usr/joe/appts'' also creates the same alias for the corresponding object. Either alias can be used to open the object.

Interobject communication is built on top of Sun RPC, which supports remote invocations of procedures, and Sun XDR, which defines machine-independent representation of data values. An invocation of a method in an object uses Sun RPC to invoke a *dispatcher* in the object, sending it a *method number* identifying the method and XDR representations of arguments to the method. The dispatcher reconstructs the machine-dependent representations of the arguments using XDR routines and invokes the specified method. In case of synchronous method invocation it also sends back a return value.

A class is compiled by an *object compiler* which processes annotations, generates support routines used for traversing data structures communicated in methods and saved and restored from the data file, and invokes the host compiler to compile the source and generated code. An object compiler is required for each language supported by the system. Our current implementation supports C.

Experience

We have used our implementation for creating prototype versions of a distributed multiuser appointment service, a distributed multiuser spreadsheet, a distributed student database, and an object-based version of an existing software testing environment called Mothra [4].

We have also used it as a basis for building our first version of SUITE (System of Uniform Interactive Type-directed Editors). In SUITE, objects combine not only the program interface of files and processes, but also their user interfaces. Like files, they can be edited, and like processes, they provide incremental response to user input. Between each object and the user is a system-provided object called a *dialogue manager*, which provides ''type-directed'' editing of selected variables of the object, and informs the object about changes to its variables by invoking methods in the object. A detailed discussion of the SUITE project is given in [7].

3. Discussion

In this section, we discuss some distinguishing properties of our approach and consequences of supporting them.

Heterogeneous Objects

Unlike Smalltalk and Eden, our approach allows objects to execute programs in multiple conventional languages. Thus the system does not force the programmer to use any one, possibly unconventional, programming language. On the other hand, single-language systems can offer an integrated programming environment where there is no distinction between system data structures and language data structures. Moreover, they can provide code sharing and classification of objects based on inheritance of classes. It is not clear how such a facility can be provided by a multilanguage operating system.

It is important to note that our approach provides object-based programming only at the operating system level. It does not address programming of the internal data structures of an object, which themselves may be objects if the programming language is object-based. We provide only a way of creating the top-level system data structures as objects. We expect these objects to be used for the same purposes for which files have been traditionally used:

- for keeping data to be shared between processes written in different programming languages and/or executing in different address spaces,
- for keeping data to be protected from access by processes executing on behalf of arbitrary users,
- for keeping persistent data.

It is conceivable that a programmer using a language that does not support objects may also create temporary, private data as top-level objects. For instance, a C programmer may create a *stack* object by defining a server that responds to **push** and **pop** messages. However, the cost of interprocess communication makes such use impractical.

Moreover, objects can be considered as serious alternatives to files only if the cost of accessing objects can compete with the cost of accessing files. Experiments show that the cost of sending messages to local and remote processes compares favourably with the cost of

accessing local and remote files [3, 11]. However, object-access has the added expense of activation and passivation of objects, which involves loading and unloading of the object's program.

We believe the above problem is not so severe since locality of object reference should make activation and passivation of objects infrequent. Moreover, a machine may not actually passivate an object when its reference count goes to zero. Instead, it may put it in a "cache" of object processes that the system looks at before creating a new object process. In case of cache hit, a previously created object process for an object can be reused.

We do not expect all system data structures to be created as objects. For instance, data that has no structure or semantics, such as some mail messages, can be kept in files, and directory information can be kept in directories. Files, directories, devices, ports, and other resources provided by the operating system can be considered as special objects whose operations are predefined and implemented by the operating system.

Hybrid System

Our approach has adopted the philosophy of extending conventional systems to support objects. Other systems, such as HYDRA, Smalltalk, Eden, and Clouds have taken the opposite approach of creating new systems based entirely on the concept of objects (that is, all system structures are created as objects). Our approach allows a programmer to experiment with objects without sacrificing a familiar domain. It also creates a system that strictly enhances the existing one. Pure object-based systems, on the other hand, are prototype kernels instead of full systems and do not offer substitutes for the large number of facilities offered by conventional systems.

One disadvantage of our approach is that it is not as uniform as pure object-based systems. Some system data structures are created as objects, while others are created as files, directories, and devices. We have tried to reduce this problem by making objects look like processes and files. In this respect, our approach is similar to the one taken by hybrid programming languages such as C++ [14] and Objective-C [5], which have tried to make objects look like conventional data structures. However, like these languages, we believe our system is not as uniform as its pure counterparts.

Objects as Files

Our exercise of integrating objects in a conventional system has lead to us giving objects several properties not supported by current object-based systems. Like files, objects

- have a hierarchical name in a network file system,
- need to be opened before access and closed after access,
- are addressed by temporary, object descriptors, instead of persistent capabilities,
- are protected by access lists instead of capability lists,
- are explicitly deleted instead of garbage collected.

We believe these properties are worth exploring even in pure object-based systems. A object-based system may support a protection scheme based on access lists because it is simple and has been successfully used in conventional systems. It may support **open_object** and **close_object** for efficiency reasons: The access rights of a client needs to be checked only when it invokes the **open_object** operation and not every time it sends a message to the client. Invocations of **close_object** can be used to decide when an object should be deactivated.

On the other hand these properties have some negative consequences. Since descriptors are temporary, object needs to go through the overhead of establishing active and passive

invariants, and need to have permanents name in a directory. These properties are unacceptable in small objects such as integers, since it would be impractical to require, for instance, that each integer have a file name, and descriptors be allocated for all integers accessed by an object. Therefore integrated programming environments that provide a uniform object model cannot support file properties in objects. However, in systems that do distinguish between small and large objects, we believe it is worth exploring these properties in large objects.

Interestingly, the notion of accessing active entities as files has also been explored in Version 8 Unix [10], which puts processes in the file system, giving them names based on their process identifiers. A debugger can use these names to open and close processes on its machine and read and write their code segments. A process in this system is not an object in that it is not persistent or protected from messages sent by arbitrary processes.

Our approach can be combined with Version 8 Unix. The Version 8 **read** and **write** operations can be defined on objects to provide debugging of objects.

4. Future Work

The design and implementation of our approach assumed no changes to the kernel of the base system. With kernel support object and data files can be made special files in the system, and the object operations **open_object, close_object, create_object,** and **delete_object** can be integrated with the corresponding file operations **open, close, create,** and **delete.** Moreover, the protection scheme can be changed to better meet the needs of objects. It can, for instance, let an object divided its messages into "read", "write", and "execute" messages protected by the **read, write,** and **execute** rights respectively.

A limitation of our approach is that it does not allow evolution of an object when its class changes. When the class of an object changes, the offsets, types, and names of its persistent variables may change. In order to upgrade the object, the old version of its persistent state needs to be mapped to the new version. We plan to support "transducers" that define the mapping between the old and new versions of a class and can be applied to an instance of the class to upgrade it.

Finally, we plan to use our implementation to explore replacement of existing Unix files and applications with corresponding objects.

5. Conclusions

Keeping system data in files instead of objects leads to at least three problems:

- Files are untyped.
- The syntax and semantics of information stored in files is hard to change.
- The in-core and external copy of data may get inconsistent.

We have presented a simple approach for supporting objects in a conventional system. In comparison to previous approaches, our approach supports multiple languages, a hybrid system, and gives objects file properties. We have implemented our approach on Unix and are currently extending it and exploring its usefulness.

Acknowledgement

Paul Thomas built an early version of the object manager.

REFERENCES

[1] G.T. Almes, A.P Black, E.D. Lazowska, and J.D. Noe, "The Eden System: A Technical Overview," *IEEE Transactions on Software Engineering SE-11*, January 1985, pp. 43-59.

[2] F. Baskett, J.H. Howard, and J.T. Montague, "Task Communication in DEMOS," *Proceedings of the Sixth Symposium on Operating System Principles*, November 1975, pp. 23-32.

[3] Bharat Bhargava, Tom Mueller, and John Riedl, "Experimental Analysis of Layered Ethernet Software," *Proceedings of the Fall Joint Computer Science Conference*, October 1987, pp. 559-568.

[4] B. J. Choi, R. A. DeMillo, E. W. Krauser, R. J. Martin, A. P. Mathur, A. J. Offutt, H. Pan, and E. H. Spafford, "The Mothra Tool Set," *Proceedings of the Hawaii International Conference on System Sciences*, January 1989.

[5] Brad Cox, *Object-Oriented Programming, An Evolutionary Approach*, Addison-Wesley, 1986.

[6] P. Dasgupta, R. J. LeBlanc, and W. A. Appelbe, "The Clouds Distributed Operating System: Functional Description, Implementation Details, and Related Work," *Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose*, June 1988, pp. 2-9.

[7] Prasun Dewan and Eric Vasilik, "The Suite Research Project," Technical Report SERC-TR-26-P, Purdue-Florida Software Engineering Research Center.

[8] R. A. Finkel, M. L. Scott, W. K. Kalsow, Y. Artsy, H-Y Chang, P. Dewan, A. J. Gordon, B. Rosenburg, M. H. Solomon, and C-Q Yang, "Experience with Charlotte: Simplicity versus function in a distributed operating system," Computer Sciences Technical Report #653, University of Wisconsin–Madison, July 1986.

[9] Michael B. Jones and Richard F. Rashid, "Mach and MatchMaker: Kernel and Language Support for Object-Oriented Distributed Systems," *OOPSLA '86 Proceedings*, September 1986, pp. 67-77.

[10] T. Killian, "Processes as Files," *Proceedings of the Summer Usenix Conference*, July 1984.

[11] Edward D. Lazowska, John Zahorjan, David Cheriton, and Willy Zwaenepoel, "File Access Performance of Diskless Workstations," *ACM Transactions on Computer Systems* 4:3 (August 1986), pp. 238-268.

[12] Barbara Liskov and Robert Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *Proceedings of the 9th POPL*, 1982, pp. 7-19.

[13] Bruce Jay Nelson, ''Remote Procedure Call,'' Tech Report CMU-CS-81-119, Department of Computer Science, Carnegie-Mellon University, May 1981.

[14] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading, Mass., 1986.

[15] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, ''Hydra: The Kernel of a Multiprocessor Operating System,'' *CACM* 17:6 (June 1974).

A Tour of the Worm

Donn Seeley

Department of Computer Science
University of Utah

ABSTRACT

On the evening of November 2, 1988, a self-replicating program was released upon the Internet[1]. This program (a *worm*) invaded VAX and Sun-3 computers running versions of Berkeley UNIX, and used their resources to attack still more computers[2]. Within the space of hours this program had spread across the U.S., infecting hundreds or thousands of computers and making many of them unusable due to the burden of its activity. This paper provides a chronology for the outbreak and presents a detailed description of the internals of the worm, based on a C version produced by decompiling.

1. Introduction

There is a fine line between helping administrators protect their systems and providing a cookbook for bad guys. [Grampp and Morris, "UNIX Operating System Security"]

November 3, 1988 is already coming to be known as Black Thursday. System administrators around the country came to work on that day and discovered that their networks of computers were laboring under a huge load. If they were able to log in and generate a system status listing, they saw what appeared to be dozens or hundreds of "shell" (command interpreter) processes. If they tried to kill the processes, they found that new processes appeared faster than they could kill them. Rebooting the computer seemed to have no effect—within minutes after starting up again, the machine was overloaded by these mysterious processes.

These systems had been invaded by a *worm*. A worm is a program that propagates itself across a network, using resources on one machine to attack other machines. (A worm is not quite the same as a *virus*, which is a program fragment that inserts itself into other programs.) The worm had taken advantage of lapses in security on systems that were running 4.2 or 4.3 BSD UNIX or derivatives like SunOS. These lapses allowed it to connect to machines across a network, bypass their login authentication, copy itself and then proceed to attack still more machines. The massive system load was generated by multitudes of worms trying to propagate the epidemic.

The Internet had never been attacked in this way before, although there had been plenty of speculation that an attack was in store. Most system administrators were unfamiliar with the concept of worms (as opposed to viruses, which are a major affliction of the PC world) and it took some time before they were able to establish what was going on and how to deal with it. This paper is intended to let people know exactly what happened and how it came about, so that they will be better prepared when it happens the next time. The behavior of the worm will be examined in detail, both to show exactly what it did and didn't do, and to show the dangers of future

[1] The Internet is a logical network made up of many physical networks, all running the IP class of network protocols.

[2] VAX and Sun-3 are models of computers built by Digital Equipment Corp. and Sun Microsystems Inc., respectively. UNIX is a Registered Bell of AT&T Trademark Laboratories.

worms. The epigraph above is now ironic, for the author of the worm used information in that paper to attack systems. Since the information is now well known, by virtue of the fact that thousands of computers now have copies of the worm, it seems unlikely that this paper can do similar damage, but it is definitely a troubling thought. Opinions on this and other matters will be offered below.

2. Chronology

> Remember, when you connect with another computer, you're connecting to every computer that computer has connected to. [Dennis Miller, on NBC's *Saturday Night Live*]

> Here is the gist of a message I got: I'm sorry. [Andy Sudduth, in an anonymous posting to the TCP-IP list on behalf of the author of the worm, 11/3/88]

Many details of the chronology of the attack are not yet available. The following list represents dates and times that we are currently aware of. Times have all been rendered in Pacific Standard Time for convenience.

11/2: 1800 (approx.)
This date and time were seen on worm files found on *prep.ai.mit.edu*, a VAX 11/750 at the MIT Artificial Intelligence Laboratory. The files were removed later, and the precise time was lost. System logging on *prep* had been broken for two weeks. The system doesn't run accounting and the disks aren't backed up to tape: a perfect target. A number of "tourist" users (individuals using public accounts) were reported to be active that evening. These users would have appeared in the session logging, but see below.

11/2: 1824 First known West Coast infection: *rand.org* at Rand Corp. in Santa Monica.

11/2: 1904 *csgw.berkeley.edu* is infected. This machine is a major network gateway at UC Berkeley. Mike Karels and Phil Lapsley discover the infection shortly afterward.

11/2: 1954 *mimsy.umd.edu* is attacked through its *finger* server. This machine is at the University of Maryland College Park Computer Science Department.

11/2: 2000 (approx.)
Suns at the MIT AI Lab are attacked.

11/2: 2028 First *sendmail* attack on mimsy.

11/2: 2040 Berkeley staff figure out the *sendmail* and *rsh* attacks, notice *telnet* and *finger* peculiarities, and start shutting these services off.

11/2: 2049 *cs.utah.edu* is infected. This VAX 8600 is the central Computer Science Department machine at the University of Utah. The next several entries follow documented events at Utah and are representative of other infections around the country.

11/2: 2109 First *sendmail* attack at *cs.utah.edu*.

11/2: 2121 The load average on *cs.utah.edu* reaches 5. The "load average" is a system-generated value that represents the average number of jobs in the run queue over the last minute; a load of 5 on a VAX 8600 noticeably degrades response times, while a load over 20 is a drastic degradation. At 9 PM, the load is typically between 0.5 and 2.

11/2: 2141 The load average on *cs.utah.edu* reaches 7.

11/2: 2201 The load average on *cs.utah.edu* reaches 16.

11/2: 2206 The maximum number of distinct runnable processes (100) is reached on *cs.utah.edu*; the system is unusable.

11/2: 2220 Jeff Forys at Utah kills off worms on *cs.utah.edu*. Utah Sun clusters are infected.

11/2: 2241 Re-infestation causes the load average to reach 27 on *cs.utah.edu*.

11/2: 2249 Forys shuts down *cs.utah.edu*.

11/3: 2321 Re-infestation causes the load average to reach 37 on *cs.utah.edu*, despite continuous efforts by Forys to kill worms.

11/2: 2328 Peter Yee at NASA Ames Research Center posts a warning to the TCP-IP mailing list: "We are currently under attack from an Internet VIRUS. It has hit UC Berkeley, UC San Diego, Lawrence Livermore, Stanford, and NASA Ames." He suggests turning off *telnet*, *ftp*, *finger*, *rsh* and SMTP services. He does not mention *rexec*. Yee is actually at Berkeley working with Keith Bostic, Mike Karels and Phil Lapsley.

11/3: 0034 At another's prompting, Andy Sudduth of Harvard anonymously posts a warning to the TCP-IP list: "There may be a virus loose on the internet." This is the first message that (briefly) describes how the *finger* attack works, describes how to defeat the SMTP attack by rebuilding *sendmail*, and explicitly mentions the *rexec* attack. Unfortunately Sudduth's message is blocked at *relay.cs.net* while that gateway is shut down to combat the worm, and it does not get delivered for almost two days. Sudduth acknowledges authorship of the message in a subsequent message to TCP-IP on Nov. 5.

11/3: 0254 Keith Bostic sends a fix for *sendmail* to the newsgroup comp.bugs.4bsd.ucb-fixes and to the TCP-IP mailing list. These fixes (and later ones) are also mailed directly to important system administrators around the country.

11/3: early morning
The *wtmp* session log is mysteriously removed on *prep.ai.mit.edu*.

11/3: 0507 Edward Wang at Berkeley figures out and reports the *finger* attack, but his message doesn't come to Mike Karels' attention for 12 hours.

11/3: 0900 The annual Berkeley Unix Workshop commences at UC Berkeley. 40 or so important system administrators and hackers are in town to attend, while disaster erupts at home. Several people who had planned to fly in on Thursday morning are trapped by the crisis. Keith Bostic spends much of the day on the phone at the Computer Systems Research Group offices answering calls from panicked system administrators from around the country.

11/3: 1500 (approx.)
The team at MIT Athena calls Berkeley with an example of how the *finger* server bug works.

11/3: 1626 Dave Pare arrives at Berkeley CSRG offices; disassembly and decompiling start shortly afterward using Pare's special tools.

11/3: 1800 (approx.)
The Berkeley group sends out for calzones. People arrive and leave; the offices are crowded, there's plenty of excitement. Parallel work is in progress at MIT Athena; the two groups swap code.

11/3: 1918 Keith Bostic posts a fix for the *finger* server.

11/4: 0600 Members of the Berkeley team, with the worm almost completely disassembled and largely decompiled, finally take off for a couple hours' sleep before returning to the workshop.

11/4: 1236 Theodore Ts'o of Project Athena at MIT publicly announces that MIT and Berkeley have completely disassembled the worm.

11/4: 1700 (approx.)
A short presentation on the worm is made at the end of the Berkeley UNIX Workshop.

11/8: National Computer Security Center meeting to discuss the worm. There are about 50 attendees.

11/11: 0038 Fully decompiled and commented worm source is installed at Berkeley.

3. Overview

What exactly did the worm do that led it to cause an epidemic? The worm consists of a 99-line bootstrap program written in the C language, plus a large relocatable object file that comes in VAX and Sun-3 flavors. Internal evidence showed that the object file was generated from C sources, so it was natural to decompile the binary machine language into C; we now have over 3200 lines of commented C code which recompiles and is mostly complete. We shall start the tour of the worm with a quick overview of the basic goals of the worm, followed by discussion in depth of the worm's various behaviors as revealed by decompilation.

The activities of the worm break down into the categories of attack and defense. Attack consists of locating hosts (and accounts) to penetrate, then exploiting security holes on remote systems to pass across a copy of the worm and run it. The worm obtains host addresses by examining the system tables */etc/hosts.equiv* and */.rhosts*, user files like *.forward* and *.rhosts*, dynamic routing information produced by the *netstat* program, and finally randomly generated host addresses on local networks. It ranks these by order of preference, trying a file like */etc/hosts.equiv* first because it contains names of local machines that are likely to permit unauthenticated connections. Penetration of a remote system can be accomplished in any of three ways. The worm can take advantage of a bug in the *finger* server that allows it to download code in place of a finger request and trick the server into executing it. The worm can use a "trap door" in the *sendmail* SMTP mail service, exercising a bug in the debugging code that allows it to execute a command interpreter and download code across a mail connection. If the worm can penetrate a local account by guessing its password, it can use the *rexec* and *rsh* remote command interpreter services to attack hosts that share that account. In each case the worm arranges to get a remote command interpreter which it can use to copy over, compile and execute the 99-line bootstrap. The bootstrap sets up its own network connection with the local worm and copies over the other files it needs, and using these pieces a remote worm is built and the infection procedure starts over again.

Defense tactics fall into three categories: preventing the detection of intrusion, inhibiting the analysis of the program, and authenticating other worms. The worm's simplest means of hiding itself is to change its name. When it starts up, it clears its argument list and sets its zeroth argument to *sh*, allowing it to masquerade as an innocuous command interpreter. It uses *fork*() to change its process I.D., never staying too long at one I.D. These two tactics are intended to disguise the worm's presence on system status listings. The worm tries to leave as little trash lying around as it can, so at start-up it reads all its support files into memory and deletes the tell-tale filesystem copies. It turns off the generation of *core* files, so if the worm makes a mistake, it doesn't leave evidence behind in the form of core dumps. The latter tactic is also designed to block analysis of the program—it prevents an administrator from sending a software signal to the worm to force it to dump a core file. There are other ways to get a core file, however, so the worm carefully alters character data in memory to prevent it from being extracted easily. Copies of disk files are encoded by repeatedly exclusive-or'ing a ten-byte code sequence; static strings are encoded byte-by-byte by exclusive-or'ing with the hexadecimal value 81, except for a private

word list which is encoded with hexadecimal 80 instead. If the worm's files are somehow captured before the worm can delete them, the object files have been loaded in such a way as to remove most non-essential symbol table entries, making it harder to guess at the purposes of worm routines from their names. The worm also makes a trivial effort to stop other programs from taking advantage of its communications; in theory a well-prepared site could prevent infection by sending messages to ports that the worm was listening on, so the worm is careful to test connections using a short exchange of random "magic numbers".

When studying a tricky program like this, it's just as important to establish what the program *does not* do as what it does do. The worm *does not delete a system's files*: it only removes files that it created in the process of bootstrapping. The program does not attempt to incapacitate a system by deleting important files, or indeed any files. It does not remove log files or otherwise interfere with normal operation other than by consuming system resources. The worm *does not modify existing files*: it is not a virus. The worm propagates by copying itself and compiling itself on each system; it does not modify other programs to do its work for it. Due to its method of infection, it can't count on sufficient privileges to be able to modify programs. The worm *does not install Trojan horses*: its method of attack is strictly active, it never waits for a user to trip over a trap. Part of the reason for this is that the worm can't afford to waste time waiting for Trojan horses—it must reproduce before it is discovered. Finally, the worm *does not record or transmit decrypted passwords*: except for its own static list of favorite passwords, the worm does not propagate cracked passwords on to new worms nor does it transmit them back to some home base. This is not to say that the accounts that the worm penetrated are secure merely because the worm did not tell anyone what their passwords were, of course—if the worm can guess an account's password, certainly others can too. The worm *does not try to capture superuser privileges*: while it does try to break into accounts, it doesn't depend on having particular privileges to propagate, and never makes special use of such privileges if it somehow gets them. The worm *does not propagate over uucp* or X.25 or DECNET or BITNET: it specifically requires TCP/IP. The worm *does not infect System V systems* unless they have been modified to use Berkeley network programs like *sendmail*, *fingerd* and *rexec*.

4. Internals

Now for some details: we shall follow the main thread of control in the worm, then examine some of the worm's data structures before working through each phase of activity.

4.1. The thread of control

When the worm starts executing in *main*(), it takes care of some initializations, some defense and some cleanup. The very first thing it does is to change its name to *sh*. This shrinks the window during which the worm is visible in a system status listing as a process with an odd name like *x9834753*. It then initializes the random number generator, seeding it with the current time, turns off core dumps, and arranges to die when remote connections fail. With this out of the way, the worm processes its argument list. It first looks for an option **–p** *$$*, where *$$* represents the process I.D. of its parent process; this option indicates to the worm that it must take care to clean up after itself. It proceeds to read in each of the files it was given as arguments; if cleaning up, it removes each file after it reads it. If the worm wasn't given the bootstrap source file *l1.c* as an argument, it exits silently; this is perhaps intended to slow down people who are experimenting with the worm. If cleaning up, the worm then closes its file descriptors, temporarily cutting itself off from its remote parent worm, and removes some files. (One of these files, */tmp/.dumb*, is never created by the worm and the unlinking seems to be left over from an earlier stage of development.) The worm then zeroes out its argument list, again to foil the system status program *ps*. The next step is to initialize the worm's list of network interfaces; these interfaces are used to find local networks and to check for alternate addresses of the current host.

Finally, if cleaning up, the worm resets its process group and kills the process that helped to bootstrap it. The worm's last act in *main*() is to call a function we named *doit*(), which contains the main loop of the worm.

doit() runs a short prologue before actually starting the main loop. It (redundantly) seeds the random number generator with the current time, saving the time so that it can tell how long it has been running. The worm then attempts its first infection. It initially attacks gateways that it found with the *netstat* network status program; if it can't infect one of these hosts, then it checks random host numbers on local networks, then it tries random host numbers on networks that are on the far side of gateways, in each case stopping if it succeeds. (Note that this sequence of attacks differs from the sequence the worm uses after it has entered the main loop.)

After this initial attempt at infection, the worm calls the routine *checkother*() to check for another worm already on the local machine. In this check the worm acts as a client to an existing worm which acts as a server; they may exchange "population control" messages, after which one of the two worms will eventually shut down.

```
doit() {
        seed the random number generator with the time
        attack hosts: gateways, local nets, remote nets
        checkother();
        send_message();
        for (;;) {
                cracksome();
                other_sleep(30);
                cracksome();
                change our process ID
                attack hosts: gateways, known hosts,
                        remote nets, local nets
                other_sleep(120);
                if 12 hours have passed,
                        reset hosts table
                if (pleasequit && nextw > 10)
                        exit(0);
        }
}
```

"C" pseudo-code for the *doit*() function

One odd routine is called just before entering the main loop. We named this routine *send_message*(), but it really doesn't send anything at all. It looks like it was intended to cause 1 in 15 copies of the worm to send a 1-byte datagram to a port on the host *ernie.berkeley.edu*, which is located in the Computer Science Department at UC Berkeley. It has been suggested that this was a feint, designed to draw attention to *ernie* and away from the author's real host. Since the routine has a bug (it sets up a TCP socket but tries to send a UDP packet), nothing gets sent at all. It's possible that this was a deeper feint, designed to be uncovered only by decompilers; if so, this wouldn't be the only deliberate impediment that the author put in our way. In any case, administrators at Berkeley never detected any process listening at port 11357 on ernie, and we found no code in the worm that listens at that port, regardless of the host.

The main loop begins with a call to a function named *cracksome*() for some password cracking. Password cracking is an activity that the worm is constantly working at in an incremental fashion. It takes a break for 30 seconds to look for intruding copies of the worm on the local host, and then goes back to cracking. After this session, it forks (creates a new process running with a copy of the same image) and the old process exits; this serves to turn over process I.D. numbers and makes it harder to track the worm with the system status program *ps*. At this point the worm goes back to its infectious stage, trying (in order of preference) gateways, hosts listed in system tables like */etc/hosts.equiv*, random host numbers on the far side of gateways and random hosts on local networks. As before, if it succeeds in infecting a new host, it marks that host in a list and leaves the infection phase for the time being. After infection, the worm spends two minutes looking for new local copies of the worm again; this is done here because a newly infected remote host may try to reinfect the local host. If 12 hours have passed and the worm is still alive, it assumes that it has had bad luck due to networks or hosts being down, and it reinitializes its table of hosts so that it can start over from scratch. At the end of the main loop the worm checks to see if it is scheduled to die as a result of its population control features, and if it is, and if it has done a sufficient amount of work cracking passwords, it exits.

4.2. Data structures

The worm maintains at least four interesting data structures, and each is associated with a set of support routines.

The *object* structure is used to hold copies of files. Files are encrypted using the function *xorbuf*() while in memory, so that dumps of the worm won't reveal anything interesting. The files are copied to disk on a remote system before starting a new worm, and new worms read the files into memory and delete the disk copies as part of their start-up duties. Each structure contains a name, a length and a pointer to a buffer. The function *getobjectbyname*() retrieves a pointer to a named object structure; for some reason, it is only used to call up the bootstrap source file.

The *interface* structure contains information about the current host's network interfaces. This is mainly used to check for local attached networks. It contains a name, a network address, a subnet mask and some flags. The interface table is initialized once at start-up time.

The *host* structure is used to keep track of the status and addresses of hosts. Hosts are added to this list dynamically, as the worm encounters new sources of host names and addresses. The list can be searched for a particular address or name, with an option to insert a new entry if no matching entry is found. Flag bits are used to indicate whether the host is a gateway, whether it was found in a system table like */etc/hosts.equiv*, whether the worm has found it impossible to attack the host for some reason, and whether the host has already been successfully infected. The bits for "can't infect" and "infected" are cleared every 12 hours, and low priority hosts are deleted, to be accumulated again later. The structure contains up to 12 names (aliases) and up to 6 distinct network addresses for each host.

In our sources, what we've called the *muck* structure is used to keep track of accounts for the purpose of password cracking. (It was awarded the name *muck* for sentimental reasons, although *pw* or *acct* might be more mnemonic.) Each structure contains an account name, an encrypted password, a decrypted password (if available) plus the home directory and personal information fields from the password file.

4.3. Population growth

The worm contains a mechanism that seems to be designed to limit the number of copies of the worm running on a given system, but beyond that our current understanding of the design goals is itself limited. It clearly does not prevent a system from being overloaded, although it does appear to pace the infection so that early copies can go undetected. It has been suggested that a simulation of the worm's population control features might reveal more about its design, and we are interested writing such a simulation.

The worm uses a client-and-server technique to control the number of copies executing on the current machine. A routine *checkother*() is run at start-up time. This function tries to connect to a server listening at TCP port 23357. The connection attempt returns immediately if no server is present, but blocks if one is available and busy; a server worm periodically runs its server code during time-consuming operations so that the queue of connections does not grow large. After the client exchanges magic numbers with the server as a trivial form of authentication, the client and the server roll dice to see who gets to survive. If the exclusive-or of the respective low bits of the client's and the server's random numbers is 1, the server wins, otherwise the client wins. The loser sets a flag *pleasequit* that eventually allows it to exit at the bottom of the main loop. If at any time a problem occurs—a read from the server fails, or the wrong magic number is returned—the client worm returns from the function, becoming a worm that never acts as a server and hence does not engage in population control. Perhaps as a precaution against a cataleptic server, a test at the top of the function causes 1 in 7 worms to skip population control. Thus the worm finishes the population game in *checkother*() in one of three states: scheduled to die after some time, with *pleasequit* set; running as a server, with the possibility of losing the game later; and immortal, safe from the gamble of population control.

A complementary routine *other_sleep*() executes the server function. It is passed a time in seconds, and it uses the Berkeley *select*() system call to wait for that amount of time accepting connections from clients. On entry to the function, it tests to see whether it has a communications port with which to accept connections; if not, it simply sleeps for the specified amount of time and returns. Otherwise it loops on *select*(), decrementing its time remaining after serving a client until no more time is left and the function returns. When the server acquires a client, it performs the inverse of the client's protocol, eventually deciding whether to proceed or to quit. *other_sleep*() is called from many different places in the code, so that clients are not kept waiting too long.

Given the worm's elaborate scheme for controlling re-infection, what led it to reproduce so quickly on an individual machine that it could swamp it? One culprit is the 1 in 7 test in *checkother*(): worms that skip the client phase become immortal, and thus don't risk being eliminated by a roll of the dice. Another source of system loading is the problem that when a worm decides it has lost, it can still do a lot of work before it actually exits. The client routine isn't even run until the newly born worm has attempted to infect at least one remote host, and even if a worm loses the roll, it continues executing to the bottom of the main loop, and even then it won't exit unless it has gone through the main loop several times, limited by its progress in cracking passwords. Finally, new worms lose all of the history of infection that their parents had, so the children of a worm are constantly trying to re-infect the parent's host, as well as the other children's hosts. Put all of these factors together and it comes as no surprise that within an hour or two after infection, a machine may be entirely devoted to executing worms.

4.4. Locating new hosts to infect

One of the characteristics of the worm is that all of its attacks are active, never passive. A consequence of this is that the worm can't wait for a user to take it over to another machine like gum on a shoe—it must search out hosts on its own.

The worm has a very distinct list of priorities when hunting for hosts. Its favorite hosts are gateways; the *hg*() routine tries to infect each of the hosts it believes to be gateways. Only when all of the gateways are known to be infected or infection-proof does the worm go on to other hosts. *hg*() calls the *rt_init*() function to get a list of gateways; this list is derived by running the *netstat* network status program and parsing its output. The worm is careful to skip the loopback device and any local interfaces (in the event that the current host is a gateway); when it finishes, it randomizes the order of the list and adds the first 20 gateways to the host table to speed up the initial searches. It then tries each gateway in sequence until it finds a host that can be infected, or it runs out of hosts.

After taking care of gateways, the worm's next priority is hosts whose names were found in a scan of system files. At the start of password cracking, the files */etc/hosts.equiv* (which contains names of hosts to which the local host grants user permissions without authentication) and */.rhosts* (which contains names of hosts from which the local host permits remote privileged logins) are examined, as are all users' *.forward* files (which list hosts to which mail is forwarded from the current host). These hosts are flagged so that they can be scanned earlier than the rest. The *hi*() function is then responsible for attacking these hosts.

When the most profitable hosts have been used up, the worm starts looking for hosts that aren't recorded in files. The routine *hl*() checks local networks: it runs through the local host's addresses, masking off the host part and substituting a random value. *ha*() does the same job for remote hosts, checking alternate addresses of gateways. Special code handles the ARPAnet practice of putting the IMP number in the low host bits and the actual IMP port (representing the host) in the high host bits. The function that runs these random probes, which we named *hack_netof*(), seems to have a bug that prevents it from attacking hosts on local networks; this may be due to our own misunderstanding, of course, but in any case the check of hosts from system files should be sufficient to cover all or nearly all of the local hosts anyway.

Password cracking is another generator of host names, but since this is handled separately from the usual host attack scheme presented here, it will be discussed below with the other material on passwords.

4.5. Security holes

> The first fact to face is that Unix was not developed with security, in any realistic sense, in mind... [Dennis Ritchie, "On the Security of Unix"]

This section discusses the TCP services used by the worm to penetrate systems. It's a touch unfair to use the quote above when the implementation of the services we're about to discuss was distributed by Berkeley rather than Bell Labs, but the sentiment is appropriate. For a long time the balance between security and convenience on Unix systems has been tilted in favor of convenience. As Brian Reid has said about the break-in at Stanford two years ago: "Programmer convenience is the antithesis of security, because it is going to become intruder convenience if the programmer's account is ever compromised." The lesson from that experience seems to have been forgotten by most people, but not by the author of the worm.

4.5.1. Rsh and rexec

> These notes describe how the design of TCP/IP and the 4.2BSD implementation allow users on untrusted and possibly very distant hosts to masquerade as users on trusted hosts. [Robert T. Morris, "A Weakness in the 4.2BSD Unix TCP/IP Software"]

Rsh and *rexec* are network services which offer remote command interpreters. *Rexec* uses password authentication; *rsh* relies on a "privileged" originating port and permissions files. Two vulnerabilities are exploited by the worm—the likelihood that a remote machine that has an account for a local user will have the same password as the local account, allowing penetration through *rexec*, and the likelihood that such a remote account will include the local host in its *rsh* permissions files. Both of these vulnerabilities are really problems with laxness or convenience for users and system administrators rather than actual bugs, but they represent avenues for infection just like inadvertent security bugs.

The first use of *rsh* by the worm is fairly simple: it looks for a remote account with the same name as the one that is (unsuspectingly) running the worm on the local machine. This test is part of the standard menu of hacks conducted for each host; if it fails, the worm falls back upon *finger*, then *sendmail*. Many sites, including Utah, already were protected from this trivial attack by not providing remote shells for pseudo-users like *daemon* or *nobody*.

A more sophisticated use of these services is found in the password cracking routines. After a password is successfully guessed, the worm immediately tries to penetrate remote hosts associated with the broken account. It reads the user's *.forward* file (which contains an address to which mail is forwarded) and *.rhosts* file (which contains a list of hosts and optionally user names on those hosts which are granted permission to access the local machine with *rsh* bypassing the usual password authentication), trying these hostnames until it succeeds. Each target host is attacked in two ways. The worm first contacts the remote host's *rexec* server and sends it the account name found in the *.forward* or *.rhosts* files followed by the guessed password. If this fails, the worm connects to the local *rexec* server with the local account name and uses that to contact the target's *rsh* server. The remote *rsh* server will permit the connection provided the name of the local host appears in either the */etc/hosts.equiv* file or the user's private *.rhosts* file.

Strengthening these network services is far more problematic than fixing *finger* and *sendmail*, unfortunately. Users don't like the inconvenience of typing their password when logging in on a trusted local host, and they don't want to remember different passwords for each of the many hosts they may have to deal with. Some of the solutions may be worse than the disease—for example, a user who is forced to deal with many passwords is more likely to write them down somewhere.

4.5.2. Finger

> *gets* was removed from our [C library] a couple days ago. [Bill Cheswick at AT&T Bell Labs Research, private communication, 11/9/88]

Probably the neatest hack in the worm is its co-opting of the TCP *finger* service to gain entry to a system. *Finger* reports information about a user on a host, usually including things like the user's full name, where their office is, the number of their phone extension and so on. The Berkeley[3] version of the *finger* server is a really trivial program: it reads a request from the originating host, then runs the local *finger* program with the request as an argument and ships the output back. Unfortunately the *finger* server reads the remote request with *gets*(), a standard C library routine that dates from the dawn of time and which does not check for overflow of the server's 512 byte request buffer on the stack. The worm supplies the finger server with a request that is 536 bytes long; the bulk of the request is some VAX machine code that asks the system to execute the command interpreter *sh*, and the extra 24 bytes represent just enough data to write over the server's stack frame for the main routine. When the main routine of the server exits, the

[3] Actually, like much of the code in the Berkeley distribution, the *finger* server was contributed from elsewhere; in this case, it appears that MIT was the source.

calling function's program counter is supposed to be restored from the stack, but the worm wrote over this program counter with one that points to the VAX code in the request buffer. The program jumps to the worm's code and runs the command interpreter, which the worm uses to enter its bootstrap.

Not surprisingly, shortly after the worm was reported to use this feature of *gets*(), a number of people replaced all instances of *gets*() in system code with sensible code that checks the length of the buffer. Some even went so far as to remove *gets*() from the library, although the function is apparently mandated by the forthcoming ANSI C standard[4]. So far no one has claimed to have exercised the finger server bug before the worm incident, but in May 1988, students at UC Santa Cruz apparently penetrated security using a different finger server with a similar bug. The system administrator at UCSC noticed that the Berkeley finger server had a similar bug and sent mail to Berkeley, but the seriousness of the problem was not appreciated at the time (Jim Haynes, private communication).

One final note: the worm is meticulous in some areas but not in others. From what we can tell, there was no Sun-3 version of the *finger* intrusion even though the Sun-3 server was just as vulnerable as the VAX one. Perhaps the author had VAX sources available but not Sun sources?

4.5.3. Sendmail

> [T]he trap door resulted from two distinct 'features' that, although innocent by themselves, were deadly when combined (kind of like binary nerve gas). [Eric Allman, personal communication, 11/22/88]

The *sendmail* attack is perhaps the least preferred in the worm's arsenal, but in spite of that one site at Utah was subjected to nearly 150 *sendmail* attacks on Black Thursday. *Sendmail* is the program that provides the SMTP mail service on TCP networks for Berkeley UNIX systems. It uses a simple character-oriented protocol to accept mail from remote sites. One feature of *sendmail* is that it permits mail to be delivered to processes instead of mailbox files; this can be used with (say) the *vacation* program to notify senders that you are out of town and are temporarily unable to respond to their mail. Normally this feature is only available to recipients. Unfortunately a little loophole was accidentally created when a couple of earlier security bugs were being fixed—if *sendmail* is compiled with the *DEBUG* flag, and the sender at runtime asks that *sendmail* enter debug mode by sending the *debug* command, it permits senders to pass in a command sequence instead of a user name for a recipient. Alas, most versions of *sendmail* are compiled with *DEBUG*, including the one that Sun sends out in its binary distribution. The worm mimics a remote SMTP connection, feeding in */dev/null* as the name of the sender and a carefully crafted string as the recipient. The string sets up a command that deletes the header of the message and passes the body to a command interpreter. The body contains a copy of the worm bootstrap source plus commands to compile and run it. After the worm finishes the protocol and closes the connection to *sendmail*, the bootstrap will be built on the remote host and the local worm waits for its connection so that it can complete the process of building a new worm.

Of course this is not the first time that an inadvertent loophole or "trap door" like this has been found in sendmail, and it may not be the last. In his Turing Award lecture, Ken Thompson said: "You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.)" In fact, as Eric Allman says, "[Y]ou can't even trust code that you did totally create yourself." The basic problem of trusting system programs is not one that is easy to solve.

---

[4] See for example Appendix B, section 1.4 of the second edition of *The C Programming Language* by Kernighan and Ritchie.

4.6. Infection

The worm uses two favorite routines when it decides that it wants to infect a host. One routine that we named *infect*() is used from host scanning routines like *hg*(). *infect*() first checks that it isn't infecting the local machine, an already infected machine or a machine previously attacked but not successfully infected; the "infected" and "immune" states are marked by flags on a host structure when attacks succeed or fail, respectively. The worm then makes sure that it can get an address for the target host, marking the host immune if it can't. Then comes a series of attacks: first by *rsh* from the account that the worm is running under, then through *finger*, then through *sendmail*. If *infect*() fails, it marks the host as immune.

The other infection routine is named *hul*() and it is run from the password cracking code after a password has been guessed. *hul*(), like *infect*(), makes sure that it's not re-infecting a host, then it checks for an address. If a potential remote user name is available from a *.forward* or *.rhosts* file, the worm checks it to make sure it is reasonable—it must contain no punctuation or control characters. If a remote user name is unavailable the worm uses the local user name. Once the worm has a user name and a password, it contacts the *rexec* server on the target host and tries to authenticate itself. If it can, it proceeds to the bootstrap phase; otherwise, it tries a slightly different approach—it connects to the local *rexec* server with the local user name and password, then uses this command interpreter to fire off a command interpreter on the target machine with *rsh*. This will succeed if the remote host says it trusts the local host in its */etc/hosts.equiv* file, or the remote account says it trusts the local account in its *.rhosts* file. *hul*() ignores *infect*()'s "immune" flag and does not set this flag itself, since *hul*() may find success on a per-account basis that *infect*() can't achieve on a per-host basis.

Both *infect*() and *hul*() use a routine we call *sendworm*() to do their dirty work[5]. *sendworm*() looks for the *ll.c* bootstrap source file in its objects list, then it uses the *makemagic*() routine to get a communication stream endpoint (a *socket*), a random network port number to rendezvous at, and a magic number for authentication. (There is an interesting side effect to *makemagic*()—it looks for a usable address for the target host by trying to connect to its TCP *telnet* port; this produces a characteristic log message from the *telnet* server.) If *makemagic*() was successful, the worm begins to send commands to the remote command interpreter that was started up by the immediately preceding attack. It changes its directory to an unprotected place (*/usr/tmp*), then it sends across the bootstrap source, using the UNIX stream editor *sed* to parse the input stream. The bootstrap source is compiled and run on the remote system, and the worm runs a routine named *waithit*() to wait for the remote bootstrap to call back on the selected port.

The bootstrap is quite simple. It is supplied the address of the originating host, a TCP port number and a magic number as arguments. When it starts, it unlinks itself so that it can't be detected in the filesystem, then it calls *fork*() to create a new process with the same image. The old process exits, permitting the originating worm to continue with its business. The bootstrap reads its arguments then zeroes them out to hide them from the system status program; then it is ready to connect over the network to the parent worm. When the connection is made, the bootstrap sends over the magic number, which the parent will check against its own copy. If the parent accepts the number (which is carefully rendered to be independent of host byte order), it will send over a series of filenames and files which the bootstrap writes to disk. If trouble occurs, the bootstrap removes all these files and exits. Eventually the transaction completes, and the bootstrap calls up a command interpreter to finish the job.

---

[5] One minor exception: the *sendmail* attack doesn't use *sendworm*() since it needs to handle the SMTP protocol in addition to the command interpreter interface, but the principle is the same.

In the meantime, the parent in *waithit*() spends up to two minutes waiting for the bootstrap to call back; if the bootstrap fails to call back, or the authentication fails, the worm decides to give up and reports a failure. When a connection is successful, the worm ships all of its files across followed by an end-of-file indicator. It pauses four seconds to let a command interpreter start on the remote side, then it issues commands to create a new worm. For each relocatable object file in the list of files, the worm tries to build an executable object; typically each file contains code for a particular make of computer, and the builds will fail until the worm tries the proper computer type. If the parent worm finally gets an executable child worm built, it sets it loose with the **–p** option to kill the command interpreter, then shuts down the connection. The target host is marked ''infected''. If none of the objects produces a usable child worm, the parent removes the detritus and *waithit*() returns an error indication.

When a system is being swamped by worms, the */usr/tmp* directory can fill with leftover files as a consequence of a bug in *waithit*(). If a worm compile takes more than 30 seconds, resynchronization code will report an error but *waithit*() will fail to remove the files it has created. On one of our machines, 13 MB of material representing 86 sets of files accumulated over 5.5 hours.

4.7. Password cracking

A password cracking algorithm seems like a slow and bulky item to put in a worm, but the worm makes this work by being persistent and efficient. The worm is aided by some unfortunate statistics about typical password choices. Here we discuss how the worm goes about choosing passwords to test and how the UNIX password encryption routine was modified.

4.7.1. Guessing passwords

> For example, if the login name is ''abc'', then ''abc'', ''cba'', and ''abcabc'' are excellent candidates for passwords. [Grampp and Morris, ''UNIX Operating System Security'']

The worm's password guessing is driven by a little 4-state machine. The first state gathers password data, while the remaining states represent increasingly less likely sources of potential passwords. The central cracking routine is called *cracksome*(), and it contains a switch on each of the four states.

The routine that implements the first state we named *crack_0*(). This routine's job is to collect information about hosts and accounts. It is only run once; the information it gathers persists for the lifetime of the worm. Its implementation is straightforward: it reads the files */etc/hosts.equiv* and */.rhosts* for hosts to attack, then reads the password file looking for accounts. For each account, the worm saves the name, the encrypted password, the home directory and the user information fields. As a quick preliminary check, it looks for a *.forward* file in each user's home directory and saves any host name it finds in that file, marking it like the previous ones.

We unimaginatively called the function for the next state *crack_1*(). *crack_1*() looks for trivially broken passwords. These are passwords which can be guessed merely on the basis of information already contained in the password file. Grampp and Morris report a survey of over 100 password files where between 8 and 30 percent of all passwords were guessed using just the literal account name and a couple of variations. The worm tries a little harder than this: it checks the null password, the account name, the account name concatenated with itself, the first name (extracted from the user information field, with the first letter mapped to lower case), the last name, and the account name reversed. It runs through up to 50 accounts per call to *cracksome*(), saving its place in the list of accounts and advancing to the next state when it runs out of accounts to try.

The next state is handled by *crack_2*(). In this state the worm compares a list of favorite passwords, one password per call, with all of the encrypted passwords in the password file. The

list contains 432 words, most of which are real English words or proper names; it seems likely that this list was generated by stealing password files and cracking them at leisure on the worm author's home machine. A global variable *nextw* is used to count the number of passwords tried, and it is this count (plus a loss in the population control game) that controls whether the worm exits at the end of the main loop—*nextw* must be greater than 10 before the worm can exit. Since the worm normally spends 2.5 minutes checking for clients over the course of the main loop and calls *cracksome*() twice in that period, it appears that the worm must make a minimum of 7 passes through the main loop, taking more than 15 minutes[6]. It will take at least 9 hours for the worm to scan its built-in password list and proceed to the next state.

The last state is handled by *crack_3*(). It opens the UNIX online dictionary */usr/dict/words* and goes through it one word at a time. If a word is capitalized, the worm tries a lower-case version as well. This search can essentially go on forever: it would take something like four weeks for the worm to finish a typical dictionary like ours.

When the worm selects a potential password, it passes it to a routine we called *try_password*(). This function calls the worm's special version of the UNIX password encryption function *crypt*() and compares the result with the target account's actual encrypted password. If they are equal, or if the password and guess are the null string (no password), the worm saves the cleartext password and proceeds to attack hosts that are connected to this account. A routine we called *try_forward_and_rhosts*() reads the user's *.forward* and *.rhosts* files, calling the previously described *hul*() function for each remote account it finds.

4.7.2. Faster password encryption

> The use of encrypted passwords appears reasonably secure in the absence of serious attention of experts in the field. [Morris and Thompson, "Password Security: A Case History"]

Unfortunately some experts in the field have been giving serious attention to fast implementations of the UNIX password encryption algorithm. UNIX password authentication works without putting any readable version of the password onto the system, and indeed works without protecting the encrypted password against reading by users on the system. When a user types a password in the clear, the system encrypts it using the standard *crypt*() library routine, then compares it against a saved copy of the encrypted password. The encryption algorithm is meant to be basically impossible to invert, preventing the retrieval of passwords by examining only the encrypted text, and it is meant to be expensive to run, so that testing guesses will take a long time. The UNIX password encryption algorithm is based on the Federal Data Encryption Standard (DES). Currently no one knows how to invert this algorithm in a reasonable amount of time, and while fast DES encoding chips are available, the UNIX version of the algorithm is slightly perturbed so that it is impossible to use a standard DES chip to implement it.

Two problems have been mitigating against the UNIX implementation of DES. Computers are continually increasing in speed—current machines are typically several times faster than the machines that were available when the current password scheme was invented. At the same time, ways have been discovered to make software DES run faster. UNIX passwords are now far more

---

[6] For those mindful of details: The first call to *cracksome*() is consumed reading system files. The worm must spend at least one call to *cracksome*() in the second state attacking trivial passwords. This accounts for at least one pass through the main loop. In the third state, *cracksome*() tests one password from its list of favorites on each call; the worm will exit if it lost a roll of the dice and more than ten words have been checked, so this accounts for at least six loops, two words on each loop for five loops to reach 10 words, then another loop to pass that number. Altogether this amounts to a minimum of 7 loops. If all 7 loops took the maximum amount of time waiting for clients, this would require a minimum of 17.5 minutes, but the 2-minute check can exit early if a client connects and the server loses the challenge, hence 15.5 minutes of waiting time plus runtime overhead is the minimum lifetime. In this period a worm will attack at least 8 hosts through the host infection routines, and will try about 18 passwords for each account, attacking more hosts if accounts are cracked.

susceptible to persistent guessing, particularly if the encrypted passwords are already known. The worm's version of the UNIX *crypt*() routine ran more than 9 times faster than the standard version when we tested it on our VAX 8600. While the standard *crypt*() takes 54 seconds to encrypt 271 passwords on our 8600 (the number of passwords actually contained in our password file), the worm's *crypt*() takes less than 6 seconds.

The worm's *crypt*() algorithm appears to be a compromise between time and space: the time needed to encrypt one password guess versus the substantial extra table space needed to squeeze performance out of the algorithm. Curiously, one performance improvement actually saves a little space. The traditional UNIX algorithm stores each bit of the password in a byte, while the worm's algorithm packs the bits into two 32-bit words. This permits the worm's algorithm to use bit-field and shift operations on the password data, which is immensely faster. Other speedups include unrolling loops, combining tables, precomputing shifts and masks, and eliminating redundant initial and final permutations when performing the 25 applications of modified DES that the password encryption algorithm uses. The biggest performance improvement comes as a result of combining permutations: the worm uses expanded arrays which are indexed by groups of bits rather than the single bits used by the standard algorithm. Matt Bishop's fast version of *crypt*() does all of these things and also precomputes even more functions, yielding twice the performance of the worm's algorithm but requiring nearly 200 KB of initialized data as opposed to the 6 KB used by the worm and the less than 2 KB used by the normal *crypt*().

How can system administrators defend against fast implementations of *crypt*()? One suggestion that has been introduced for foiling the bad guys is the idea of shadow password files. In this scheme, the encrypted passwords are hidden rather than public, forcing a cracker to either break a privileged account or use the host's CPU and (slow) encryption algorithm to attack, with the added danger that password test requests could be logged and password cracking discovered. The disadvantage of shadow password files is that if the bad guys somehow get around the protections for the file that contains the actual passwords, all of the passwords must be considered cracked and will need to be replaced. Another suggestion has been to replace the UNIX DES implementation with the fastest available implementation, but run it 1000 times or more instead of the 25 times used in the UNIX *crypt*() code. Unless the repeat count is somehow pegged to the fastest available CPU speed, this approach merely postpones the day of reckoning until the cracker finds a faster machine. It's interesting to note that Morris and Thompson measured the time to compute the old M-209 (non-DES) password encryption algorithm used in early versions of UNIX on the PDP-11/70 and found that a good implementation took only 1.25 milliseconds per encryption, which they deemed insufficient; currently the VAX 8600 using Matt Bishop's DES-based algorithm needs 11.5 milliseconds per encryption, and machines 10 times faster than the VAX 8600 at a cheaper price will be available soon (if they aren't already!).

5. Opinions

> The act of breaking into a computer system has to have the same social stigma as breaking into a neighbor's house. It should not matter that the neighbor's door is unlocked. [Ken Thompson, 1983 Turing Award Lecture]

> [Creators of viruses are] stealing a car for the purpose of joyriding. [R H Morris, in 1983 Capitol Hill testimony, cited in the New York Times 11/11/88]

I don't propose to offer definitive statements on the morality of the worm's author, the ethics of publishing security information or the security needs of the UNIX computing community, since people better (and less) qualified than I are still copiously flaming on these topics in the various network newsgroups and mailing lists. For the sake of the mythical ordinary system administrator who might have been confused by all the information and misinformation, I will try to answer a few of the most relevant questions in a narrow but useful way.

Did the worm cause damage? The worm did not destroy files, intercept private mail, reveal passwords, corrupt databases or plant Trojan horses. It did compete for CPU time with, and eventually overwhelm, ordinary user processes. It used up limited system resources such as the open file table and the process text table, causing user processes to fail for lack of same. It caused some machines to crash by operating them close to the limits of their capacity, exercising bugs that do not appear under normal loads. It forced administrators to perform one or more reboots to clear worms from the system, terminating user sessions and long-running jobs. It forced administrators to shut down network gateways, including gateways between important nation-wide research networks, in an effort to isolate the worm; this led to delays of up to several days in the exchange of electronic mail, causing some projects to miss deadlines and others to lose valuable research time. It made systems staff across the country drop their ongoing hacks and work 24-hour days trying to corner and kill worms. It caused members of management in at least one institution to become so frightened that they scrubbed all the disks at their facility that were online at the time of the infection, and limited reloading of files to data that was verifiably unmodified by a foreign agent. It caused bandwidth through gateways that were still running after the infection started to become substantially degraded—the gateways were using much of their capacity just shipping the worm from one network to another. It penetrated user accounts and caused it to appear that a given user was disturbing a system when in fact they were not responsible. It's true that the worm could have been far more harmful that it actually turned out to be: in the last few weeks, several security bugs have come to light which the worm could have used to thoroughly destroy a system. Perhaps we should be grateful that we escaped incredibly awful consequences, and perhaps we should also be grateful that we have learned so much about the weaknesses in our systems' defenses, but I think we should share our gratefulness with someone other than the worm's author.

Was the worm malicious? Some people have suggested that the worm was an innocent experiment that got out of hand, and that it was never intended to spread so fast or so widely. We can find evidence in the worm to support and to contradict this hypothesis. There are a number of bugs in the worm that appear to be the result of hasty or careless programming. For example, in the worm's *if_init*() routine, there is a call to the block zero function *bzero*() that incorrectly uses the block itself rather than the block's address as an argument. It's also possible that a bug was responsible for the ineffectiveness of the population control measures used by the worm. This could be seen as evidence that a development version of the worm "got loose" accidentally, and perhaps the author originally intended to test the final version under controlled conditions, in an environment from which it would not escape. On the other hand, there is considerable evidence that the worm was designed to reproduce quickly and spread itself over great distances. It can be argued that the population control hacks in the worm are anemic by design: they are a compromise between spreading the worm as quickly as possible and raising the load enough to be detected and defeated. A worm will exist for a substantial amount of time and will perform a substantial amount of work even if it loses the roll of the (imaginary) dice; moreover, 1 in 7 worms become immortal and can't be killed by dice rolls. There is ample evidence that the worm was designed to hamper efforts to stop it even after it was identified and captured. It certainly succeeded in this, since it took almost a day before the last mode of infection (the *finger* server) was identified, analyzed and reported widely; the worm was very successful in propagating itself during this time even on systems which had fixed the *sendmail* debug problem and had turned off *rexec*. Finally, there is evidence that the worm's author deliberately introduced the worm to a foreign site that was left open and welcome to casual outside users, rather ungraciously abusing this hospitality. He apparently further abused this trust by deleting a log file that might have revealed information that could link his home site with the infection. I think the innocence lies in the research community rather than with the worm's author.

Will publication of worm details further harm security? In a sense, the worm itself has solved that problem: it has published itself by sending copies to hundreds or thousands of

machines around the world. Of course a bad guy who wants to use the worm's tricks would have to go through the same effort that we went through in order to understand the program, but then it only took us a week to completely decompile the program, so while it takes fortitude to hack the worm, it clearly is not greatly difficult for a decent programmer. One of the worm's most effective tricks was advertised when it entered—the bulk of the *sendmail* hack is visible in the log file, and a few minutes' work with the sources will reveal the rest of the trick. The worm's fast password algorithm could be useful to the bad guys, but at least two other faster implementations have been available for a year or more, so it isn't very secret, or even very original. Finally, the details of the worm have been well enough sketched out on various newsgroups and mailing lists that the principal hacks are common knowledge. I think it's more important that we understand what happened, so that we can make it less likely to happen again, than that we spend time in a futile effort to cover up the issue from everyone but the bad guys. Fixes for both source and binary distributions are widely available, and anyone who runs a system with these vulnerabilities needs to look into these fixes immediately, if they haven't done so already.

6. Conclusion

> It has raised the public awareness to a considerable degree. [R H Morris, quoted in the New York Times 11/5/88]

This quote is one of the understatements of the year. The worm story was on the front page of the New York Times and other newspapers for days. It was the subject of television and radio features. Even the *Bloom County* comic strip poked fun at it.

Our community has never before been in the limelight in this way, and judging by the response, it has scared us. I won't offer any fancy platitudes about how the experience is going to change us, but I will say that I think these issues have been ignored for much longer than was safe, and I feel that a better understanding of the crisis just past will help us cope better with the next one. Let's hope we're as lucky next time as we were this time.

Acknowledgments

No one is to blame for the inaccuracies herein except me, but there are plenty of people to thank for helping to decompile the worm and for helping to document the epidemic. Dave Pare and Chris Torek were at the center of the action during the late night session at Berkeley, and they had help and kibitzing from Keith Bostic, Phil Lapsley, Peter Yee, Jay Lepreau and a cast of thousands. Glenn Adams and Dave Siegel provided good information on the MIT AI Lab attack, while Steve Miller gave me details on Maryland, Jeff Forys on Utah, and Phil Lapsley, Peter Yee and Keith Bostic on Berkeley. Bill Cheswick sent me a couple of fun anecdotes from AT&T Bell Labs. Jim Haynes gave me the run-down on the security problems turned up by his busy little undergrads at UC Santa Cruz. Eric Allman, Keith Bostic, Bill Cheswick, Mike Hibler, Jay Lepreau, Chris Torek and Mike Zeleznik provided many useful review comments. Thank you all, and everyone else I forgot to mention.

Matt Bishop's paper "A Fast Version of the DES and a Password Encryption Algorithm", ©1987 by Matt Bishop and the Universities Space Research Association, was helpful in (slightly) parting the mysteries of DES for me. Anyone wishing to understand the worm's DES hacking had better look here first. The paper is available with Bishop's *deszip* distribution of software for fast DES encryption. The latter was produced while Bishop was with the Research Institute for Advanced Computer Science at NASA Ames Research Center; Bishop is now at Dartmouth College (*bishop@bear.dartmouth.edu*). He sent me a very helpful note on the worm's implementation of *crypt*() which I leaned on heavily when discussing the algorithm above.

The following documents were also referenced above for quotes or for other material:

Data Encryption Standard, FIPS PUB 46, National Bureau of Standards, Washington D.C., January 15, 1977.

F. T. Grampp and R. H. Morris, "UNIX Operating System Security," in the *AT&T Bell Laboratories Technical Journal*, October 1984, Vol. 63, No. 8, Part 2, p. 1649.

Brian W. Kernighan and Dennis Ritchie, *The C Programming Language*, Second Edition, Prentice Hall: Englewood Cliffs, NJ, ©1988.

John Markoff, "Author of computer 'virus' is son of U.S. Electronic Security Expert," p. 1 of the *New York Times*, November 5, 1988.

John Markoff, "A family's passion for computers, gone sour," p. 1 of the *New York Times*, November 11, 1988.

Robert Morris and Ken Thompson, "Password Security: A Case History," dated April 3, 1978, in the *UNIX Programmer's Manual*, in the *Supplementary Documents* or the *System Manager's Manual*, depending on where and when you got your manuals.

Robert T. Morris, "A Weakness in the 4.2BSD Unix TCP/IP Software," AT&T Bell Laboratories Computing Science Technical Report #117, February 25, 1985. This paper actually describes a way of spoofing TCP/IP so that an untrusted host can make use of the *rsh* server on any 4.2 BSD UNIX system, rather than an attack based on breaking into accounts on trusted hosts, which is what the worm uses.

Brian Reid, "Massive UNIX breakins at Stanford," RISKS-FORUM Digest, Vol. 3, Issue 56, September 16, 1986.

Dennis Ritchie, "On the Security of UNIX," dated June 10, 1977, in the same manual you found the Morris and Thompson paper in.

Ken Thompson, "Reflections on Trusting Trust," 1983 ACM Turing Award Lecture, in the *Communications of the ACM*, Vol. 27, No. 8, p. 761, August 1984.

Some Musings on Ethics and Computer Break-Ins

Eugene H. Spafford

Department of Computer Sciences
Purdue University
W. Lafayette, IN 47907-2004

spaf@cs.purdue.edu

ABSTRACT

In November and December, the computing community experienced the release of the Internet Worm, computer break-ins at Lawrence Livermore National Labs, and the temporary disconnection of the Milnet because of computer break-ins on a machine belonging to the Mitre Corporation. These incidents have led to many discussions about responsibility and ethics. Many of these discussions, particularly in forums such as the Usenet, have become heated without leading to any commonly-accepted conclusions.

This paper addresses some of these points. The intent is to summarize a few of the principal arguments supporting various positions and to argue some points of particular merit. At the end, references are given to material that may help provide background material for readers seeking further information.

Included in this discussion are the questions of whether individuals breaking into our machines are doing us a favor, and whether those individuals should in any way be encouraged. The paper concludes with some observations about the importance of the discussion, and the need to reach a consensus in the computer profession, if not in society as a whole.

Introduction

On November 2, a program was run on the Internet that replicated itself on thousands of machines, often loading them to the point where they were unable to process normal requests. This *Internet Worm* program was stopped in a matter of hours, but the controversy engendered by its release has raged for weeks. The occurrence of other machine break-ins, possibly as "copycat" acts, has only served to fuel these discussions.

It is important that we discuss these issues. The continuing evolution of our technological base and our increasing reliance on computers for critical tasks implies that future incidents may well be more serious in their consequences. We must also admit that there will be more such incidents, with human nature as varied and extreme as it is, and with the technology as available as it is.

This paper presents a brief introduction to a few of the major issues that have been raised by the Worm incident. My intent is to present some of the arguments on each of these topics as a basis for further thought and discussion. References are provided at the end of the paper to enable the interested reader to explore particular aspects of the problem in greater depth.

In the following discussion, I have separated issues that often have been combined when debated. This, by itself, may help to clarify some of the issues; it is possible that most people are in agreement on some of these points once they are viewed as individual issues.

A Comment on Ethics

Ethics is the study of what is *right* to do. Alternatively, it is sometimes described as the study of what is *good* and how to achieve that good. To enable me to suggest whether an act is right or wrong, it is necessary for me to describe an ethical system that you, the reader, can understand and apply. I have made the simplifying assumption that we can judge the ethical nature of an act by applying a Kantian assessment: would we view that act as proper if **everyone** were to engage in it on a regular basis? Although I am aware that this may be too simplistic a model (and it could be argued that other ethical philosophies can also be applied), I will use this as a first approximation for purposes of argument. Kant's work has considerably influenced our contemporary ethical thought, and using this approach provides an understandable, initial point of reference; the reader unfamiliar with the philosophy and science of ethics may wish to consult some of the cited references for further information.

Note that this philosophy assumes that *right* is determined at least as much by actions as by results. Some ethical philosophies assume that "the ends justify the means." Our current society does not operate by such a philosophy, although many individuals do. As a society we profess to believe that "it isn't whether you win or lose, it's how you play the game." This is why we are concerned with issues of due process and civil rights, even for those individuals espousing repugnant views.

In summary, to judge the ethical good of an activity by the results rather than the means to achieve those results is not in keeping with our current social mores. Also, it is important to note that philosophies that consider the results as the measure of good are often impossible to apply because of the difficulty of understanding exactly what results from any arbitrary activity.

Consider an (extreme) example: if the government were to behead, on live TV, 1000 people chosen at random because they smoke, then the result might well be that many hundreds of thousands of smokers would quit "cold turkey," thus prolonging their lives. It might also prevent hundreds of thousands of people from ever starting to smoke, thus improving the health and longevity of the general populace. Despite the great good this would hold for society, everyone except for a few extremists would condemn such an idea as immoral. We would likely object even if only one person were involved. It would not matter what the law might be on such a matter; we would not feel that it was morally correct.

Note that we would be unable to judge the morality of such an act by evaluating the results, because we would not know the full scope of those results. Such an act might have effects on issues of law, public health, tobacco use, and daytime TV shows for decades or centuries to follow. Such a system of ethics would not allow us to evaluate our current activities at the time we would need such guidance.

Issues and Discussion

One of the first issues to be discussed after the release of the Internet Worm dealt with the intent of the author. A common topic of speculation was exactly why the Worm program had been written and released to the Internet. Suggested explanations put forth by the community have ranged from "simple accident" to "the actions of a hostile sociopath."

The problem with any speculation on this topic is that we do not **know** the real intent. Although an author has been alleged, he has not made any public statement about his reasons; because of pending legal actions, it is unlikely that such a statement will be made anytime soon.

Thus, any suppositions about his intent must be viewed as little better than guesswork.

One speculation can be addressed, however. Some individuals contend that the release of the program was an accident—that it somehow "escaped" during test. There is one compelling reason that suggests that this is not the case: the program had no way of being stopped, as would be be the case if it were being tested. The code contained no check that would allow someone to halt the program, nor was there any mechanism to limit the spread off the machine. The code did not check for any file or other condition that would signal it should stop. Rather, it contained code to evaluate with caution and sometimes ignore the one condition that was to limit its growth—the presence of another Worm on the same machine. Once started, the program would spread unchecked. For these reasons, it seems implausible that the code was under test. Thus, the conclusion must be that the Worm was started on purpose—it was a deliberate act.

It has been argued that the Worm was released to illustrate security defects to a community that would not otherwise pay attention. This seems unlikely. The alleged author was well-known to personnel at many universities and major companies, and his talents were generally respected. Had he merely explained the problems or offered a demonstration to these people, he would have been listened to with considerable attention. If we consider the immediate response he obtained in October when he revealed the bug in *ftp*, there appears to be little merit in the arguments that he did not know how to report a bug or that no one would listen to him.

In the more general case, this argument is also without merit. Although some system administrators might have been complacent about the security of their systems before this incident, personnel at Berkeley, most computer vendors, managers of government computer installations, and system admininstrators at major universities and colleges have always been attentive to reports of security problems. Someone wishing to report a problem with the security of a system need not exploit it to report it. It is incumbent on someone concerned about a problem to report it to a proper authority. (One does not set fire to the neighborhood shopping center to bring attention to a fire hazard inherent in the parking lot, then try to justify the act by claiming that the firemen never listen when anyone calls.)

It is worth noting that some people have advanced the related argument that bugs and security flaws will not get fixed without drastic measures. The claim is that the people who break into systems are performing a service by illustrating these problems, and they should be rewarded. This argument is severely flawed in several ways. First, it assumes that there is some compelling need to force users to install security fixes on their systems, and thus *computer burglars* are justified in engaging in "breaking and entering" activities. Taken to extremes, it suggests that it would be perfectly acceptable to engage in such activities on a continuing basis, so long as they might expose security flaws. This completely loses sight of the purpose of the computers in the first place—to serve as tools and resources, not as exercises in security. The same reasoning would imply that vigilantes have the right to attempt to break into the homes in my neighborhood on a continuing basis to demonstrate that they are susceptible to burglars. When considered from this perspective, coupled with our working definition of *right*, we can see that it would be immoral to reward computer burglars and vandals.

Another flaw with this argument is that it completely ignores the technical and economic causes that prevent many sites from upgrading or correcting their software. Not every site has the resources to install new system software or to correct existing software. At many sites, the systems are run as turnkey[1] systems—employed as tools, and maintained by the vendor. The

[1] So named because the users expect to purchase the systems, plug them in, "turn the key," and use them. This is common in business environments where the employees are using a fixed set of applications.

owners and users of these machines simply do not have the ability to correct or maintain their systems independently, and they are unable to afford custom software support from their vendors. To break into such systems, with or without damage, is to trespass into their places of business; to do so in a vigilante effort to somehow force them to upgrade their security structure is presumptuous and reprehensible.

A related contention is that the users themselves are somehow responsible for such break-ins if they are not running the latest security enhancements and corrections. Again, this assumes that there is some moral or legal responsibility to make such patches, and that every user has the capability to make such patches. *Computer burglars* break into systems by choice, not because of the presence or absence of some particular software. A burglary is not justified, morally or legally, by an argument that the victim has poor locks and was therefore "asking for it."

A further argument has been made that vendors are responsible for the on-going maintenance of their software, and that such security breaches should immediately require vendors to issue corrections to their customers, past and present. This is not economically feasible, nor is it technically workable. Many sites customize their software or otherwise run systems incompatible with the latest vendor releases. For a vendor to be able to provide instant response to security problems, it would be necessary for each customer to run completely standardized software and hardware mixes to ensure the correctness of vendor-supplied updates. Not only would this be considerably less attractive for many customers, but the increased cost of such "instant" fix distribution would add to the price of any such system—increasing the cost borne by the customer. It is unreasonable to expect the user community[2] to sacrifice flexibility **and** pay a higher cost per unit just for faster corrections to the occasional security breach.

Vendors do bear some responsibility for security, however. They supply a product to their customers that should meet some minimum expected standards of fitness. For the average user, this includes an expectation that the data on the system will be secure. Vendors should therefore be sure to test the software with security in mind. Further, the vendors should always educate their customers about the extent to which they can trust the security of their systems, and to understand the support (if any) the vendor will provide for security-related problems.

Favor?

An oft-quoted comment was that the author of the Internet Worm did us a favor by exposing the security flaws. A favor is defined as doing someone a kindness. In a sense, the Worm as constructed was a kindness—it could have been written to be destructive instead of just contagious. As it was, the worm only exposed some serious security flaws, and it heightened our awareness of security concerns. It was in this sense a kindness, although it is not something for which we should thank its author. The result does not make the activity morally correct; as has already been explained, if everyone were to write and run such programs, it would clearly be wrong.

The complete results of this incident are not yet clear. There have been other break-ins since the release of the Worm, perhaps inspired by the media coverage of the Worm. More attempts may yet be made. Some sites on the Internet have restricted access to their machines, and others may be removed from the network (or already have been). Combined with the many decades of person-hours devoted to cleaning up after the Worm, this seems to be a high price to pay for a "favor."

---

[2] Especially the UNIX community.

The legal consequences of this act are also not yet known. For instance, a bill has been introduced into the House of Representatives, most likely in response to this incident. HR-5061, entitled "The Computer Virus Eradication Act of 1988," introduced in the House by Wally Herger (R-CA) and Robert Carr (D-MI), may be only the first in a series of legislative actions that have the potential to significantly affect the computer profession. In particular, HR-5061 is notable since its wording prevents it from being applied to true computer viruses[3], and its provisions might be interpreted in such a way that almost any distributed software would violate the act. The passage of this bill or similar well-intentioned but misinformed legislation could have a negative effect on the computing profession as a whole.

Conclusions

The principal argument made in this paper is that computer break-ins, even when no obvious damage results, are unethical. This must be the considered conclusion even if the result is an improvement in security, since the activity itself is disruptive and immoral. The results of the act should be considered separately from the act itself. This is especially true when we consider how difficult it is to understand all the effects resulting from such an act.

Historically, computer professionals as a group have not been overly concerned with questions of ethics and propriety. Individuals and some organizations have tried to address these issues, but they cannot be resolved in any comprehensive way without the participation of the whole computing community. Too often, we view computers simply as machines and algorithms, and we do not perceive the ethical questions inherent in their use. When we stop to consider, however, that these machines influence the quality of life of millions of individuals, both directly and indirectly, we come to understand that there are broader issues. Computers are used to design, analyze, support, and control applications that protect and guide the lives and finances of people. Our use (and misuse) of computing systems may have effects beyond our wildest imagining. It is with this in mind that we must reconsider our attitudes about acts demonstrating a lack of respect for the rights and privacy of other people's computers and data.

We must also consider what our attitudes will be towards future security problems. In particular, we should consider the effect of **widely** publishing the source for worms, viruses and other threats to security. We know there are sites where users will be unable or unwilling to install updates and corrections. Widespread publication of detailed code will only serve to aid computer burglars to break into systems such as those.[4] Publication should serve a useful purpose; endangering the security of other people's machines or attempting to force them into making changes they are unable to make or afford is not ethical. At the same time, we need to develop a method of rapidly developing and disseminating corrections and enhancements as they become known.

Finally, we must decide these issues of ethics as a community of professionals and then present them to society as a whole. No matter what laws are passed, and no matter how good security measures might become, they will not be enough for us to have completely secure systems. We also need to develop and act according to some shared ethical values. The members of society need to be educated so that they understand the importance of respecting the privacy and ownership of data. If locks and laws were all that kept people from robbing houses, there would be many more burglars than there are now; the shared social mores about the sanctity of

[3] It provides penalties only in cases where **programs** are introduced into computer systems, not sections of code or subroutines.

[4] To anticipate the oft-used comment that the "bad guys" already have such information: not every computer burglar knows or will know every system weakness—unless we provide them with detailed analyses.

personal property are a major and important influence in the prevention of burglary. It is our duty as informed professionals to help extend those mores into the computer age.

Suggested Readings

Background information on the *Internet Worm* incident may be found in [Seel89]. and [Spaf89] An introduction to personal and professional ethics can be found in [John85]. [Tayl75] is a good general introduction to various systems of ethics. [Winc68] relates the philosophy of ethics to the development of personal morals. [Lyon84] relates ethics to law. Immanuel Kant's *Foundations of the Metaphysics of Morals* has been translated numerous times, but is probably not a good reference for any but the most serious inquiries.

There are many fine works relating to computer ethics and professional ethics. I would recommend the books [Hoff82], [John85], and [Mart89] as initial references. Other references worth reading include: [Absh81], [Absh82], [Bloo88], [Chri88], [Mitc86], and [Weiz86].

Many works have been published dealing with computer burglars and hackers. Interesting references include [Bair87], [Good83], and especially [Lee86]. The articles [Reid86] and [Stol88] may also provide some insight.

The social consequences of tampering with computers can be inferred from works such as [Hoff86] and the continuing feature "Risks to the Public" present in all recent issues of *ACM SIGSOFT Software Engineering News*. Issues of *ACM SIGCAS Computers and Society* usually contain interesting articles about the effects of computers on society, too.

I believe **everyone** involved with computer systems, especially individuals concerned with issues of trust and security, should read [Thom84].

References

Absh81.
Abshire, Gary M., "Ethics Apply to Computer Specialists, Too," *COMPUTERS AND SOCIETY*, vol. 11, no. 2, p. 22, University of Wisconsin, Madison, Wisconsin, Spring 1981.

Absh82.
Abshire, Gary M., "The ethical insensitivity of computer specialists and what you can do about it," *COMPUTERS AND SOCIETY*, vol. 12, pp. 10-11, ACM Special Interest Group on Computers and Society, Winter 1982.

Bair87.
Baird, Bruce J., Lindsay L. Baird Jr., and Ronald P. Ranauro, "The Moral Cracker?," *COMPUTERS AND SECURITY*, vol. 6, no. 6, pp. 471-478, North-Holland, December 1987.

Bloo88.
Bloombecker, J. J. Buck, "Computer Ethics for Cynics," *COMPUTERS AND SOCIETY*, vol. 18, no. 3, pp. 30-32, ACM Special Interest Group on Computers and Society, New York, NY, July 1988.

Chri88.
Christiansen, Donald, "A Matter Of Ethics," *IEEE SPECTRUM*, vol. 25, p. 15, August, 1988.

Good83.
Goodfellow, Geoffrey S., "Testimony by Geoffrey S. Goodfellow," *SOFTWARE ENGINEERING NOTES*, vol. 8, pp. 18-23, 1983.

Hoff86.
Hoffman, Lance J. and Lucy M. Moran, "Societal Vulnerability to Computer System Failures," *COMPUTERS AND SECURITY*, vol. 5, no. 3, Elsevier Science Publishers B.V.,

North-Holland, 1986.

Hoff82.
Hoffman, W. Michael and Jennifer Mills Moore (eds.), *Ethics and the Management of Computer Technology,* Oelgeschlager, Gunn & Hain, Cambridge, Mass., 1982.

John85.
Johnson, Deborah G. and John W. Snapper, *Ethical Issues in the Use of Computers,* Wadsworth Publishing Co., Belmont, CA, 1985. out of print

Lee86.Lee, John A. N., Gerald Segal, and Rosalie Steier, "Positive Alternatives: A Report on an ACM Panel on Hacking," *COMMUNICATIONS OF THE ACM*, vol. 29, no. 4, pp. 297-299, ACM, New York, NY, April, 1986.

Lyon84.
Lyons, David, in *ETHICS AND THE RULE OF LAW*, Cambridge University Press, Bath, Great Britain, 1984.

Mart89.
Martin, Mike W. and Roland Schinzinger, *Ethics in Engineering,* McGraw Hill, 1989. 2nd. Edition

Mitc86.
Mitcham, Carl, "Computers: From Ethos and Ethics to Mythos and Religion, Notes on the New Frontier Between Computers and Philosophy," *TECHNOLOGY IN SOCIETY, AN INTERNATIONAL JOURNAL*, vol. 8, no. 3, Pergamon Press, New York, 1986.

Reid86.
Reid, Brian, "Lessons from the UNIX Breakins at Stanford," *SOFTWARE ENGINEERING NOTES*, vol. 11, no. 5, pp. 29-35, ACM, October 1986.

Seel89.
Seeley, Donn, "A Tour of the Worm," *PROCEEDINGS OF 1989 WINTER USENIX CONFERENCE*, Usenix Association, San Diego, CA, February 1989.

Spaf89.
Spafford, Eugene H., "The Internet Worm Program: An Analysis," *COMPUTER COMMUNICATION REVIEW*, vol. 19, no. 1, ACM SIGCOM, January 1989. Also issued as Purdue CS technical report TR-CSD-823

Stol88.
Stoll, Clifford, "Stalking the Wily Hacker," *COMMUNICATIONS OF THE ACM*, vol. 31, no. 5, pp. 484-497, ACM, New York, NY, May 1988.

Tayl75.
Taylor, Paul W., *Principles of Ethics,* Dickenson Publishing Company, Inc., Encino, CA, 1975.

Thom84.
Thompson, Ken, "Reflections on Trusting Trust," *COMMUNICATIONS OF THE ACM*, vol. 27, no. 8, pp. 761-763, ACM, New York, NY, August 1984.

Weiz86.
Weizenbaum, Joseph, "Not Without Us," *ACM SIGCAS: COMPUTERS & SOCIETY*, vol. 16, no. 2,3, pp. 2-7, Association for Computing Machinery, Inc., New York, NY, Summer/Fall 1986.

Winc68.
Winch, Peter, *Moral Integrity*, Oxford, England, 1968.

A Comparison of Compiler Utilization of Instruction Set Architectures

Daniel V. Klein

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15217
dvk@sei.cmu.edu
+1 412 268 7791

ABSTRACT

This paper compares the utilization of a number of different computer instruction sets by a collection of compilers. Wherever possible, several compilers were used for each architecture. This paper demonstrates that CISC instruction sets are underutilized by compilers, while RISC instruction sets are nearly completely utilized. We observe that if an instruction exists on a computer, it should be usable by the compilers for that computer. Because CISC computers have large numbers of instructions which are not effectively used by compilers, the instructions are superfluous. By eliminating superfluous and redundant instructions from architectures, future systems can run more efficiently, and algorithms can be executed with greater celerity.

1. Introduction

The data reported on in this paper are the result of a Reduced Instruction Set Computer (RISC) assessment project conducted at the Software Engineering Institute.† When I entered into the RISC assessment project, it was with a strong bias toward Complex Instruction Set Computer (CISC) architectures. I was a vocal proponent of the VAX and MC680x0 architectures, and looked at this project as an interesting exercise in which I would have my (negative) suspicions about RISC processors confirmed, and one in which I would find vindication for the CISC side in the great *"RISC versus CISC"* debate.

I have, however, come to the opposite conclusion. My research on this project has convinced me (quite consistently, I might add) that, if there is a "right" side of the debate to be on, it is the RISC side. In all features – execution speed, compiler efficiency, language consistency, and code size – the concept of a *reduced* instruction set computer has proven to be the correct architectural choice.

This paper, however, does not compare benchmark statistics, nor does it contrast the execution speeds of various machines. In general, benchmarks results present highly misleading statistics, and offer little or no insight into the subtle effects that can alter benchmark performance. The performance of the Whetstone benchmark on a machine tells you little about how fast it will run *your* programs – instead, it tells you is how fast the Whetstone benchmark ran. The Dhrystone benchmark specifically requires you to disable certain compiler optimizations, a request which defeats optimizations that would otherwise show off the power of a compiler – power which can be used quite effectively in "real life" applications.

What this paper does discuss is how well compilers can use the instruction sets for which they are targeted. The purpose of this study is to ask (through demonstration) "if the great majority of programmers write in a high level language (and not assembly language), and if certain instructions cannot be used by a compiler, then why are they in the instruction set of the machine?" Of course, some instructions will never be used by a compiler, and these are not considered in our discussion. Examples of these instructions

† This work was sponsored in part by the U.S. Department of Defense.

are those used to effect context switch, return from trap or interrupt, etc. These instructions are necessary for the proper functioning of an operating system, and not for user level utilities. The "superfluous" instructions in question include the `polyf` instruction on the VAX (which calculates a floating point polynomial from a vectored set of coefficients), the redundant logical operations on the condition codes of the 680x0 (which are just as easily simulated using a simple load/store instruction and the logical operations on integers), and other overly complex instructions which no compiler can utilize.

Although not specifically called out in this paper, many CISC architectures have instructions which the compiler is sometimes able to use, but which are so "special case" that their inclusion into the architecture is quite questionable, especially when these instructions could easily be simulated with simple combinations of other simple instructions. One example of this type of instruction is the `aobleq` instruction on the VAX, which adds one to a variable, and branches to a location if the result is less than or equal to a comparison variable. This instruction is (at best) infrequently used, and its function could easily be performed by the trio of an `add`, a `cmp`, and a `bleq` instruction. A compiler (and certainly most assembly language programmers) can just as easily use a template containing 3 instructions as they can use a template containing a single instruction.

For an audience such as the Unix community, the next question which usually comes to mind is "of what use is this information to Unix systems?" Answering this question entails placing the cart before the horse, and giving some conclusions before we present our methods and results. The answer to this question is simple. A great deal of Unix systems are based on CISC architectures – the VAX and the 680x0 family are the most prevalent of these. Yet if it is indeed the case that Unix compilers† (and in fact, compilers in general) are unable to utilize the instruction sets efficiently, is there then not a great deal of "wasted silicon" – hardware which is never used, but which nonetheless takes up space and more importantly *time* – in these systems? If a RISC system can be used more efficiently than a comparably sized and priced CISC system, then not only will the users of the RISC system benefit from the increased speed of the individual instructions, but the compilers will be able to make better use of the hardware, and thus return a second benefit to the users – that of efficient (and thus, celeritous) use of the instruction set of the machine.

2. Methods

The methods used to test the instruction set utilization of the compilers was rather simple, but we believe it was effective. A collection of integer based applications were given to the compiler, and the resultant assembly code was examined. Purely integer based applications were chosen over those that also contained floating point for a number of reasons:

1) Some processors followed the IEEE 754 floating point standard, while others implemented their own floating point instructions. While following a standard is laudable, the machine architecture should not be penalized for elaborate nature of the IEEE standard.

2) Some architectures utilized a floating point co-processor. While this is a valid option, floating point instructions can either be executed as separate instructions, or as sub-codes passed to a single co-processor instruction. Evaluating these instructions in either manner can introduce a bias in the measurement techniques.

3) Integer applications abound in the Unix environment, but finding a representative floating point application is difficult. The use of *float* versus *double* variables can affect the use of one or another instruction mode, and it is very difficult to find a generic application which uses both variable types (while it is easy to find applications which mix *long* and *short* integers).

The six integer applications and libraries were chosen for their size (a larger program is more likely to use a large number of language features) and breadth (different applications do different things, and hence exercise different aspects of the instruction set). The programs chosen for the evaluation were:

† To make this a fair evaluation, we used multiple vendor's compilers for the same architecture wherever possible. For example for the VAX, our comparisons were based on the Berkeley, Tartan Labs, and DEC (VMS) C compilers. For the 68020 and SPARC, the Sun and Gnu compilers were used. By using multiple compilers, we hoped to eliminate the bias that could be introduced by a single good (or bad) compiler.

1) *csh* – The Unix C-shell, containing 15,058 lines of code,†
2) *vi* – A screen based editor, containing 24,414 lines of code,
3) *libcurses* – The Unix screen display library, containing 5,496 lines of code,
4) *dc* – An arbitrary precision arithmetic package of 2,000 lines of code,
5) *bc* – A *yacc* based front end for *dc*, containing 830 lines of human and machine generated code,
7) *efl* – An extended FORTRAN language preprocessor, containing 18,281 lines of code,
7) *libmp* – The Unix multi-precision math library, with 778 lines of math intensive code, and
8) *troff* – The Unix typesetter runoff program, containing 7853 lines of truly baroque code.

To maintain accuracy of our results, the same source code was compiled on each of the tested machines. We present the statistics for the six packages together, rather than inundating the reader with individual analyses. In truth, the compiler generated roughly the same instruction mix for each program, so we feel it is fair to present the average mix for each compiler. Although we would have liked to include other classes of programs in our analysis (e.g., an operating system, graphics tools, CAD tools, and database applications), the highly consistent output from the compilers for the applications which we did consider leads us to believe that we would have seen an insignificant difference in instruction and addressing mode usage from the set of applications selected.

In all cases, we did *not* count the instructions in the run-time libraries or the C initialization or finalization routines, since some of these are written in assembly language, and are not generated by the compiler.

For the CISC architectures, where the final machine instructions are nearly identical to the input assembly language, instructions and addressing modes are counted from the assembly language output by the C compiler. Since it would not adversely affect our statistics, the Unix `jcond` instructions were counted as a single conditional branch instruction (instead of actually seeing which resulted in a simple branch, and which were a combination of a branch-around and jump instruction). For the RISC architectures, however, where the assembler typically also performs code reorganization, the instructions and modes were counted (wherever possible) from the disassembled object files. In this latter case, where assembler reorganization could substantially affect our results, extra care was taken to insure accuracy.

For each architecture, we present three tables. The first table shows the frequency with which each class of instruction is used. In these tables, the following definitions are used: "Load" instructions fetch data from memory into registers, while "store" instructions place the contents of registers into memory locations. "Shuffle" instructions move data between registers. "Arithmetic" instructions include add, subtract, and shift, while "logical" instructions include bitwise and, or, and exclusive or. "Compare" instructions are similar to logical instructions, except that they compute single bit results (for use in subsequent conditional branch instructions). The terms "conditional branch", "unconditional branch", and "call" should be self explanatory.

In the second table we show the patterns of instruction usage – the frequency with which instructions are used irrespective of type. From this we can see the percentage of the instruction set which is used the most, and what percentage is used the least. The third table compares the use of the various addressing modes available in the architecture.

3. RISC Architectures

We first consider three RISC architectures – the MIPS R2000, the Sun SPARC, and the Motorola 88100 – which will be later compared and contrasted with the two CISC architectures – the Digital VAX and the Motorola 68020.

---

† The lines-of-code count in all cases was determined with the command

```
cc -E *.c | sed -e '/^$/d' -e '/^#/d' | wc -l
```

3.1. Analysis of the MIPS R2000 C Compiler

Tables 1 through 3 show the instruction mix generated for the six applications on the MIPS R2000. It should be noted that the instruction mix presented here is that *generated* by the compiler, and not a list of the instructions *executed* by the applications. In other words, we present a static analysis of the code generated by the compiler, rather than a dynamic one. The two concepts are fundamentally different, and for this paper we did not research the latter. However, we have no reason to doubt that the two values will be substantially the same.

Instruction Class	Count	% used
Load	30990	26.2
Store	15137	12.8
Shuffle	8350	7.1
Move	**54477**	**46.0**
Arithmetic	15890	13.4
Logical	2159	1.8
Compute	**18049**	**15.2**
Compare	2104	1.8
Conditional Branch	11609	9.8
Unconditional Branch	6482	5.5
Call	8659	7.3
No-op	16934	14.3
Control	**45788**	**38.7**
Total	**118314**	**100.00%**

Table 1 – R2000 Instruction Use

The MIPS R2000 is a classic load-store RISC architecture. Data values must be loaded into registers before they can be changed by other instructions and stored back into memory. Almost all the instructions execute in a single machine cycle, but some instructions require a delay state before data becomes valid (e.g., a load requires a delay state before the value enters memory) or before a state change (branch instructions do not complete until the second machine cycle). Many of these delay states can be filled with other instructions (in the case of a load instruction for example, any instruction which does not rely on the loaded data can be used). In those cases where no suitable instruction can be moved into the delay state, a `nop` instruction is used. This accounts for the high percentage of `nop` instructions in the count.†

Percentage use of Instructions	Number of Instructions
Never Used	12
< 0.05%	3
≤ 1.0%	20
≤ 2.5%	6
≤ 5.0%	3
≤ 7.5%	3
≤ 10.0%	1
≤ 15.0%	2
> 15.0%	1

Table 2 – R2000 Patterns of Usage

The pattern of instruction usage shown in Figure 2 shows another expected result – namely that a large percentage of the instructions are used with approximately 1% of the time (each), while a few instructions are

† The MIPS assembler reorganizer (that part of the compiler which is responsible for filling in delay slots with instructions other than `nop` instructions) is somewhat conservative in its reorganization strategy. If it were more aggressive, the percentage of `nop` instructions could be reduced.

used with great frequency. The low frequency instructions and the high frequency instructions each take up approximately half of the work load.

Of the 49 instructions on the R2000, 33 are used at least 0.05% of the time, and only 12 instructions are never used. This means that the compiler uses 73% of the instruction set, and would indicate that the compiler and the instruction set are well matched (in that the architecture is adequate for the task, and the compiler covers the instruction set).

Recall that we are not considering floating point or co-processor instructions in our count, but that we *are* considering even those instructions which are likely to be used only in an operating system context. Of the 12 instructions which the MIPS compiler does not generate, 3 are associated with operating system functions, another 3 are used only when arithmetic overflow checking is necessary (which C does not require, but other languages do), and yet another is used only when a branch target is farther from the source than any of our tests allowed. If these are eliminated from our count, the R2000 C compiler is able to use 86% of the instruction set.

For all of the RISC architectures, some leeway should be allowed in the instruction usage count. The MIPS R2000 assembly language reference guide lists a `nop` instruction, when in fact this operation is performed by a shift by zero bits of the zero register into itself. (Other non-operations exist in the instruction set, such as an add immediate to self of 0. I am told that this choice of no-op, however, presented itself as a result of the layout of the R2000 IC mask, in that it has an instruction code of 0x00000000). A `move` instruction is similarly listed as being in the instruction set, when in fact a `move` is really just an unsigned `add` with an addend of zero. The SPARC and MC88100 instructions perform similar prestidigitization.

Address Mode	Example	Count	% used
Immediate	`35`	47300	21.7
Absolute	`label`	1755	0.8†
Register	`r2`	131027	60.1
Displacement	`35(r2)`	37898	17.4
Total		**217980**	**100.0%**

Table 3 – R2000 Addressing Mode Use

Looking at the addressing modes available on the R2000, we see that all of them are used, and that all of them are used with a reasonable frequency. Although absolute addressing is infrequently used, it is essential for accessing global variables (and its function cannot be easily duplicated by any combination of any other addressing mode).

3.2. Analysis of Motorola 88100 C Compilers

The 88100 is Motorola's new RISC processor, announced in late 1988. We were able to compare two different compilers for the 88100. In this case the compilers were:

1) The Green Hills 88100 compiler (version 1.8.4), and

2) The Gnu 88100 compiler (version 1.30).

The Gnu compiler was billed as untested, but generated what appeared to be correct code. Occasionally, however, it dumped core and was unable to completely process a source file (this happened extensively in the source code for *efl*, which does all sorts of questionably legal things). Because of this, the instruction counts and addressing mode usage counts for the Gnu compiler are lower than they should be. The results obtained from the two compilers were similar, although as with other sets of compilers, the individual code idioms and instruction counts varied. Tables 4 through 6 summarize the results obtained with the two 88100 compilers.

† In order to examine the real instructions an addressing modes used on the R2000, an object code disassembler must be used. In the disassembler, Absolute mode is used only for jump instructions (including calls to subroutines). Branches and load address instructions, although coded with labels in the assembly language, are reported by the disassembler as Immediate mode operands. This accounts for the disproportionately low frequency of use of the Absolute addressing mode.

Instruction Class	Green Hills Count	Green Hills % used	Gnu Count	Gnu % used
Load	23907	22.5	24611	30.4
Store	9959	9.4	9887	12.2
Shuffle	1215	1.1	1115	1.4
Move	**35081**	**33.0**	**35613**	**44.1**
Arithmetic	12996	12.2	8386	10.4
Logical	25060	23.6	9428	11.7
Compute	**38056**	**35.8**	**17814**	**22.0**
Compare	4898	4.6	4905	6.1
Conditional Branch	11502	10.8	9288	11.5
Unconditional Branch	7893	7.4	5256	6.5
Call	8762	8.3	7958	9.8
Control	**33055**	**31.1**	**27407**	**33.9**
Total	**106192**	**100.0%**	**80834**	**100.0%**

Table 4 – 88100 Instruction Use

A familiar pattern is seen in the 88100 pattern of instruction usage. For this processor, the number of move instructions is somewhat low, and the logical instructions are rather high. One reason for this is that there is no "move" instruction on the 88100. This function is assumed by an `or` with the zero register into the destination register. If the `or` instructions used for this purpose (approximately 19.4% of the total instruction count for the Green Hills compiler and 10.3% of the total for the Gnu compiler) are counted as load instructions, the ratio of **Move : Compute : Control** instructions becomes **52.4 : 16.4 : 31.1** for the Green Hills compiler and **54.4 : 11.7 : 33.9** for the Gnu compiler, figures which are much closer to the norm we will see throughout this paper.

Unlike the R2000 and the SPARC, the 88100 does not use `nop` instructions to fill in delay slots following load operations. Instead, it uses a "scoreboard" register to keep track of which data registers are presently "in transit". It uses the contents of this register to cause delays whenever needed (e.g., when the contents of a data register is not yet valid due to a load in progress). Consequently there are no `nops` needed on the 88100. This accounts for the lower fraction of control operations in the instruction mix.

Percentage use of Instructions	Number of Instructions *Green Hills*	*Gnu*
Never Used	23	24
< 0.05%	10	6
≤ 1.0%	10	10
≤ 2.5%	4	9
≤ 5.0%	7	4
≤ 7.5%	3	4
≤ 10.0%	2	2
≤ 15.0%	2	2
> 15.0%	0	0

Table 5 – 88100 Patterns of Usage

When we look at the pattern of instruction usage for the 88100, we see the similar curve of roughly half of the instructions being used for roughly a third of the work, another 4 instructions performing roughly half of the work, and the remainder taking up the slack. This pattern repeats itself throughout most of the architectures examined (with the greatest variation being in the number of instructions which were unused).

The Green Hills compiler used 38 of the integer instructions, meaning that 62% of the instruction set (61% for the Gnu compiler) is used by the compiler. Again, this indicates that the compiler is making effective use of the instruction set. As with the R2000, some of the never used instructions are designed

for operating system use, or are used for array bounds checking. If these instructions are eliminated from our count, the percentage of instructions used rises to 70% for the the Green Hills compiler (69% for the Gnu compiler).

Address Mode	Example	*Green Hills* Count	*Green Hills* % used	*Gnu* Count	*Gnu* % used
Immediate	`35`	31207	12.4	12864	7.0
Condition†	`eq0`	9231	3.7	8715	4.8
Bit Field†	`3<0>`	826	0.3	1100	0.6
PC Relative	`label`	26663	10.6	21457	11.8
Register	`r2`	149673	59.5	103174	56.7
Register Indirect Immediate	`r2,35`	32674	13.0	24054	13.2
Register Indirect Index‡	`r2,r3`	0	—	9706	5.3
Register Indirect Scaled Index	`r2[r3]`	1192	0.5	931	0.5
Total		**251466**	**100.0%**	**182001**	**100.0%**

Table 6 – 88100 Addressing Mode Use

The addressing modes on the 88100 almost all involve registers, and syntactically appear quite similar. The differentiation is found in the instruction to which the registers are applied. The three register indirect modes are only used with the load, store, and memory exchange instructions.

Again, we see that the modes used (and as we will see later, most frequently used by complex architectures, too) are immediate, register, absolute, and some form of register indirect. In the 88100, some instructions automatically use registers as a source of an indirect address, so Register mode is counted rather high. In truth, a large fraction of this mode could be counted with Scaled mode (a special type of indirection which is explicitly called out in the assembler).

3.3. Analysis of SPARC C Compilers

The SPARC is the new "standard" architecture designed by Sun Microsystems. At least one other hardware manufacturer has adopted the SPARC instruction set architecture and is developing a GaAs version of the chip. We were able to evaluate two different compilers for the SPARC. In this case the compilers were:

1) The Sun SPARC compiler, and

2) The Gnu SPARC compiler (version 1.30).

Regrettably, the Gnu compiler was incomplete, and often dumped core during compilation (again, typically while compiling parts of *efl*). We therefore present the statistics of this compiler with the *caveat* that the results may be incomplete and inconclusive.

The results obtained from the two compilers were, however, similar. As with other sets of compilers, the individual code idioms and instruction counts varied, although the general patterns of instruction and addressing mode usage was consistent between the two compilers. Tables 7 through 9 summarize the results obtained with the two SPARC compilers.

† The Condition and Bit Field modes are really just mnemonic devices for specifying single bits or collections of bits, and are another form of Immediate operand. They are included only for completeness, but should be counted as immediate operands.

‡ The Green Hills compiler does not use Register Indirect Index mode. One reason for this is that when it needs an index of zero, it uses Register Indirect Immediate mode with an immediate operand of 0. On the other hand, the Gnu compiler accomplishes this by using Register Indirect Index with the zero register. These are essentially equivalent when the index value is zero, and counts for 13733 of the accesses with the Green Hills compiler, 8871 of the Gnu compiler.

Instruction Class	Sun Count	Sun % used	Gnu Count	Gnu % used
Load	29018	29.4	13320	19.2
Store	6822	6.9	4026	5.9
Shuffle	11717	11.9	8806	12.8
Move	**47557**	**48.2**	**26062**	**37.9**
Arithmetic	8312	8.4	5930	8.6
Logical	4354	4.4	7254	10.6
Compute	**12666**	**12.8**	**13184**	**19.2**
Compare	9866	10.0	5023	7.3
Conditional Branch	9636	9.8	5424	7.9
Unconditional Branch	4467	4.5	3465	5.0
Call	10703	10.8	7136	10.4
No-op	3779	3.8	8454	12.3
Control	**38471**	**39.0**	**29502**	**42.9**
Total	**98694**	**100.0%**	**68748**	**100.0%**

Table 7 – SPARC Instruction Use

A familiar pattern is seen in the SPARC pattern of instruction usage. For the Gnu compiler for this processor (as with both compilers for the 88100), the number of move instructions is somewhat low, and the compute instructions are rather high. One reason for this is that there is no "load address" instruction on the SPARC. This function is broken up into two instructions: `sethi`, which loads the high 22 bits of the address, and an `or` instruction which loads the low bits of the address. If the `or` instructions used for this purpose (approximately 9.5% of the total instruction count) are counted as load instructions, the ratio of **Move : Compute : Control** instructions for the Gnu compiler becomes **47.4 : 9.7 : 42.9**, which is much closer to the norm we have seen previously.

Percentage use of Instructions	Number of Instructions *Sun*	*Gnu*
Never Used	37	40
< 0.05%	5	5
≤ 1.0%	13	14
≤ 2.5%	10	6
≤ 5.0%	8	6
≤ 7.5%	1	2
≤ 10.0%	2	3
≤ 15.0%	2	2
> 15.0%	0	0

Table 8 – SPARC Patterns of Usage

The SPARC is a difficult architecture for which to classify instructions when considering the patterns of use. On the one hand, it clearly follows many of the precepts established for load-store RISC architectures. On the other hand, it has 48 different conditional branch and conditional trap instruction mnemonics (none of the latter being used by the C compiler), branch instructions which either execute or annul the following instruction on condition FALSE, and instruction mnemonics for arithmetic and logical instructions that modify the condition code register, as well as arithmetic and logical instructions that do not.

For purposes of this study, our best intuitive sense was that the paired conditional branch and branch-annul instructions should be counted as a single instruction. We also felt that the "modify condition codes" bit was an option to the various instructions (since it essentially gates the output of the condition code calculations into the condition code register) rather than altering the essential function of the instruction.

The SPARC instruction set seems to be covered very well by the two compilers, showing the same even distribution of instruction usage of the other RISC compilers. Looking at the raw numbers, though, the Sun compiler uses 41 out of 78 instructions, or 52.6% of the instruction set (48.7% for the Gnu compiler) – a rather poor showing for a RISC architecture. If, however, we eliminate the kernel and system specific instructions from consideration (including the conditional trap instructions which are never used by C but are probably used extensively by Ada), the coverage of the instruction set rises to a much more respectable 69.5% (64.4% for the Gnu compiler).

		Sun		*Gnu*	
Address Mode	**Example**	**Count**	**% used**	**Count**	**%used**
Immediate	`35`	40950	23.2	10784	11.3
Absolute	`label`	21276	12.0	19363	20.3
Register-1†	`%2`	93772	53.1	54599	57.2
Register-2	`%2+%5`	0	—	0	—
Indirect-1	`[%3]`	3718	2.1	8580	9.0
Indirect-2	`[%3+%5]`	722	0.4	0	—
Indirect-3	`[%3+37]`	16145	9.1	2083	2.2
Total		**176583**	**100.0%**	**95409**	**100.0%**

Table 9 – SPARC Addressing Mode Use

With the SPARC, we see the same 4 basic modes being used with roughly the same frequency as the other two RISC architectures. The dual register mode – Indirect-2 – is used infrequently in a static count. This mode is very useful for stepping through arrays and structures, and we feel that this mode would be used extensively in a dynamic analysis.

As shall be shown in the subsequent sections, the four addressing modes that are shared by the three RISC architectures are the same as those modes which are used most frequently by the CISC architectures we shall now examine.

4. CISC Architectures

The second part of our analysis is to compare the results of CISC compilers with those of the RISC compilers. The two CISC machines that were chosen were those found in most Unix systems currently on the market, namely the Digital VAX and the Motorola 68020.

4.1. Analysis of VAX C Compilers

The VAX is a classic CISC register architecture – almost all instructions can use almost all of the addressing modes, with many instructions having both a two and three operand format. There is no need to load the instruction operands into registers – the addressing modes can reference memory as well as register based data.

The same six applications and libraries were run through the following VAX compilers

1) Berkeley VAX C Compiler (Ultrix version 1.2),
2) Tartan Labs C Compiler (version of March 12 1986), and
3) DEC VMS C Compiler (version 2.4).

The results obtained from the three compilers were very similar. While the actual code idioms generated by the three compilers were different, and while the count of individual instructions varied somewhat, the statistics that we examined for this report are surprisingly similar. All three values are reported in Tables 10 through 12.

† On the SPARC, there are actually only 5 addressing modes. Register-1 is a special case of Register-2, where the zero register is used as the second register (and hence is not expressed in the assembler output). Similarly, Indirect-1 is a special case of Indirect-2, where the second register is the zero register. In the interests of examining all the possible permutations and their frequency of use, the special cases are separated from each other.

Instruction Class	Berkeley Count	Berkeley % used	Tartan Labs Count	Tartan Labs % used	DEC (VMS) Count	DEC (VMS) % used
Move	**26949**	**40.2**	**28217**	**40.7**	**30941**	**39.8**
Arithmetic	5359	8.0	5460	7.9	6320	8.1
Logical	804	1.2	956	1.4	1012	1.3
Compute	**6163**	**9.2**	**6416**	**9.2**	**7332**	**9.4**
Compare	8952	13.3	8734	12.6	10945	14.1
Conditional Branch	10551	15.7	10632	15.3	10894	14.0
Unconditional Branch	6016	9.0	6602	9.5	6737	8.7
Call	8473	12.6	8776	12.6	8994	11.6
No-op†	0	—	0	—	1926	2.5
Control	**33992**	**50.7**	**34744**	**50.1**	**39496**	**50.8**
Total	**67104**	**100.0%**	**69377**	**100.0%**	**77769**	**100.0%**

Table 10 – VAX Instruction Use

The first point of interest is the comparison between the VAX and the general RISC instruction usage. Although the actions of the various instructions are quite different on the two machines, the ratios of move instructions to compute and control instructions is similar on the two machines. This similarity means two things:

1) The applications and libraries used in this evaluation are a correct choice, since they produce similar results on two highly different architectures, or

2) The simple instructions on the R2000 are as adequate to the task as are the complex ones on the VAX.

To further address the second point, let us examine the frequency of instruction usage on the VAX:

Percentage use of Instructions	Number of Instructions *Berkeley*	*Tartan Labs*	*DEC (VMS)*
Never Used	117	101	111
< 0.05%	34	41	26
≤ 1.0%	44	48	54
≤ 2.5%	5	10	8
≤ 5.0%	3	5	7
≤ 7.5%	4	2	1
≤ 10.0%	0	0	1
≤ 15.0%	3	3	2
> 15.0%	0	0	0

Table 11 – VAX Patterns of Usage

Some very interesting information now presents itself. Ignoring those instructions which are never used, or used only rarely, the frequency of instruction use very closely parallels that of the R2000 instruction usage.‡ What is most notable, however, is that of the 210 integer instructions on the VAX, over 100 are *never used* by either of the three compilers, and another 33 (on the average) are used < 0.05% of the time. This means that depending on the compiler, between 65% and 72% of the integer instruction set that is available to the VAX C compilers is *never used*, or used with such a small degree of frequency as to make one ask "why are these instructions present in the architecture?"

† The DEC VMS C compiler uses `nop` instructions to cause labels (i.e., branch targets) to be placed on even byte address boundaries. Often, these `nop` instructions immediately follow a branch instruction (the branch around the "else" clause of an "if-then-else"), so that they are never executed and incur no run-time penalty. Neither the Berkeley nor the Tartan Labs compiler uses this technique.

‡ The author would hasten to point out that the ordinate values in Tables 2, 5, 8, 11, and 15 were chosen *prior* to knowing those on the abscissa. This is not a case of massaging the data to fit a curve – the data fits the curve all by itself!

In addition to the 210 integer instructions, there are an additional 110 floating point instructions which are not being considered in our calculations. To be sure, some of the instructions which the compiler never or rarely generates are used only in an operating system context, but many are what would be expected to be generated. To be fair to the designers of the VAX, some of these instructions indeed have a function. There are some instructions that are specifically designed for FORTRAN or for PL/1, but even these perform functions that could very easily be executed using one or two other instructions.

One wonderful example of this is the `ediv` instruction, which calculates both the quotient and remainder for an integer division. Unfortunately, neither the Berkeley, Tartan Labs, nor the DEC (VMS) compiler take advantage of this fact, and use this instruction only for calculating remainders (the division being performed by other means). The Minsky exception principle says that the compilers should probably have some special case processing to recognize when both a quotient and remainder are being calculated, and use the `ediv` instruction for just this purpose. The fact is, however, that either none of the compiler teams thought of this special case, or that the level of effort required to implement it was sufficiently high to warrant not including it. In either case, much the complexity of the instruction set is unused for these and similar reasons.

The VAX also has a very large number of addressing modes. The great flexibility in addressing modes is considered one of the strong selling points of CISC architectures. However, when the frequency of use of these modes is examined, some questions arise:

		Berkeley		*Tartan Labs*		*DEC (VMS)*	
Address Mode	**Example**	**Count**	**% used**	**Count**	**% used**	**Count**	**% used**
Immediate	`$270`	4685	4.6	5105	4.9	4279	3.8
Literal	`$24 (n < 64)`	14942	14.8	15122	14.5	16449	14.4
Absolute	`$*label`	0	—	0	—	0	—
Absolute Indexed	`$*label[r4]`	0	—	0	—	0	—
Relative	`label`	37067	36.7	38692	37.0	32043	28.1†
Relative Indexed	`label[r4]`	526	0.5	464	0.4	221	0.2
Relative Deferred	`*label`	337	0.3	380	0.4	0	—
Relative Deferred Indexed	`*label[r4]`	0	—	58	0.1	0	—
Register	`r3`	25480	25.3	28684	27.5	36894	32.3†
Deferred	`(r3)`	2334	2.3	3256	3.1	9552	8.4†
Deferred Indexed	`(r3)[r4]`	97	0.1	44	—	568	0.5
Autoincrement	`(r3)+`	480	0.5	632	0.6	553	0.5
Autoincrement Indexed	`(r3)+[r4]`	0	—	0	—	0	—
Deferred Autoincrement	`*(r3)+`	0	—	0	—	0	—
Deferred Autoincrement Indexed	`*(r3)+[r4]`	0	—	0	—	0	—
Autodecrement	`-(r3)`	935	0.9	1290	1.2	1076	0.9
Autodecrement Indexed	`-(r3)[r4]`	0	—	0	—	0	—
Displacement	`24(r3)`	13059	12.9	9827	9.4	11568	10.1
Displacement Indexed	`24(r3)[r4]`	86	0.1	89	0.1	167	0.1
Displacement Deferred	`*24(r3)`	753	0.7	670	0.6	568	0.5
Displacement Deferred Indexed	`*24(r3)[r4]`	122	0.1	169	0.2	123	0.1
Total		**100903**	**100.0%**	**104482**	**100.0%**	**114061**	**100.0%**

Table 12 – VAX Addressing Mode Use

Of the 21 addressing modes available on the VAX, 7 are *never used* by the Berkeley and Tartan Labs C compilers, 8 by the DEC C compiler. Another 9 are used (on the average) less than 1% of the time, so that simulating their actions through a combination of other instructions and modes would be worthwhile.

† The VMS C compiler exhibits a substantially higher percentage of Register and Deferred (i.e., indirect) mode usage than do the other two compilers. This is readily understood when one realizes that the Unix compilers use only registers R6-R11 for local (non-temporary) variables, while the VMS compiler is allowed to use registers R0-R11 for the same purpose. This extra allowance enables the VMS compiler to store addresses in registers instead of using the Relative addressing (i.e., memory direct) mode.

In fact, when we examine the addressing modes that are used most frequently (namely Immediate, Literal, Relative, Register, and Displacement), we find that the modes correspond exactly to those available on the simpler RISC instruction set of the MIPS (Immediate and Literal modes on the VAX are identical but for the length of the operand, and correspond to the MIPS Immediate mode). Considered together, the usage of these five simple modes comprises between 96.4% and 97.8% of all the addressing mode usage on the VAX!

Why then are all of these addressing modes present, if they are hardly ever used, and when they can be simulated using other modes? The VAX was supposedly designed with the help of compiler writers. With all the superfluous instructions and addressing modes, one is inclined to ask "What happened?" There can be no answer to this question – all of the fancy addressing modes of the VAX are superfluous, and need not be present in an architecture at all. Compiler technology has only gotten better over the last 10 years, so one can only conclude that these frills were never necessary.

Before we leave the VAX, let us examine one more point, namely that of the 3 operand instructions. Many VAX instructions allow the programmer to specify three operands instead of two, so that the high level instruction `a = b + c` may be coded as:

```
addl3 b, c, a
```

instead of the slightly more cumbersome:

```
movel b, a
addl2 c, a
```

Many compilers go to great length to try to use this 3 operand mode, since it results in smaller and more efficient code. However, in spite of all this effort on the part of the compilers, the 3 operand mode is greatly underutilized:

	Berkeley		*Tartan Labs*		*DEC (VMS)*	
Number of Operands	**Count**	**% used**	**Count**	**% used**	**Count**	**% used**
2 Operand	29155	91.0	30233	91.0	33884	92.0
3 Operand	2870	9.0	3004	9.0	2930	8.0
Total	**64576**	**100.0%**	**66689**	**100.0%**	**72913**	**100.0%**

Table 13 – 2 Operand *vs.* 3 Operand Addressing

As can be seen, the 3 operand mode is used (at best) only 9.0% of the time. The instruction logic necessary to distinguish between these modes is made unnecessarily complicated (and slowed) by requiring it to handle addressing modes that are rarely used. To be sure, the 2 operand mode requires that more instructions be executed. The whole machine is slowed by its complexity. If the architecture only had 1 and 2 operand modes, then the whole system would run faster (having one less mode to decode), and only in 4.5% of the instructions (at worst) would extra work be incurred. We feel that the payoff is on the side of simplicity, and not on that of complexity.

4.2. Analysis of MC68020 C Compilers

The MC68020 is another classic CISC architecture that is widely used in the industry, and especially in the Unix marketplace. It differs from the VAX in that the 68020 has specialized address and data registers, while the VAX has general registers which can be used for either purpose. Other than this primary difference, it shares with the VAX the ability to directly address both memory and registers from most instructions, and the large number of addressing modes and instruction types.

We were able to evaluate two different compilers for the 68020. In this case the compilers were:

1) The Sun 680x0 compiler (operating system version 3.5), and

2) The Gnu 680x0 compiler (version 1.31).

The results obtained from the two compilers (both with the 68020 code generation option enabled) were again similar. As with the VAX, the individual code idioms and instruction counts varied, although the general patterns of instruction and addressing mode usage (as seen in Tables 14 through 16) was

consistent across the four compilers.

Instruction Class	Sun Count	Sun % used	Gnu Count	Gnu % used
Move	**41383**	**46.3**	**39518**	**46.6**
Arithmetic	12566	14.1	10527	12.4
Logical	1386	1.6	1370	1.6
Compute	**14021**	**15.7**	**12011**	**14.2**
Compare	9281	10.4	9050	10.7
Conditional Branch	9950	11.1	10385	12.3
Unconditional Branch	5844	6.5	5561	6.6
Call	8856	9.9	8226	9.7
Control	**33931**	**38.0**	**33222**	**39.2**
Total	**89335**	**100.0%**	**84751**	**100.0%**

Table 14 – 68020 Instruction Use

Examining the instruction class coverage, we see patterns that are quite similar to the other processors and compilers in the test suite. This similarity of use of the various instruction classes adds to our belief that our methods and test suite were valid.

When we examine the pattern of instruction use, we find that as with the VAX, the 68020 compilers are unable to effectively utilize the instruction set of the target architecture.

Percentage use of Instructions	Number of Instructions *Sun*	*Gnu*
Never Used	87	89
< 0.05%	13	11
≤ 1.0%	22	24
≤ 2.5%	7	3
≤ 5.0%	7	9
≤ 7.5%	1	2
≤ 10.0%	2	1
≤ 15.0%	0	0
> 15.0%	1	1

Table 15 – 68020 Patterns of Usage

Without considering the floating point co-processor instructions, of the 140 instructions listed in the MC68020 instruction set, 88 (on the average) of the instructions are *never used* by the compilers, and another 12 (on the average) are used < 0.05% of the time. This means that roughly 71% of the 68020 instruction set is *not used* by either the Sun or the Gnu C compiler. Of course, some of these "unused" instructions are used only in an operating system context (i.e., return from trap or test-and-set instructions). Nevertheless, the very large fraction of unused or unusable instructions seems to indicate that the 68020 instruction set architecture is overly complicated. The 68030 introduces even more addressing modes – we wonder how, or even whether they will be used by compilers.

Address Mode	Example	*Berkeley* Count	*Berkeley* % used	*Gnu* Count	*Gnu* % used
Immediate	`#270`	18911	13.9	17031	13.0
Absolute	`label`	40042	29.4	35352	27.0
Register	`d3`	48901	35.9	52905	40.4
Register Indirect	`a3@`	5449	4.0	8740	6.7
Postincrement	`a3@+`	510	0.4	1229	0.9
Predecrement	`a3@-`	6062	4.5	8019	6.1
Displacement	`a3@(4)`	14952	11.0	6351	4.8
PC Displacement	`pc@(4)`	0	—	0	—
Indexed	`a3@(4,d3:w:2)`	1076	0.8	1237	0.9
PC Indexed	`pc@(4,d3:w:2)`	152	0.1	150	0.1
Memory Postindex	`([4,a3],d3:w:2,300)`	0	—	0	—
Memory Preindex	`([4,a3,d3:w:2],300)`	0	—	0	—
PC Memory Postindex	`([4,pc],d3:w:2,300)`	0	—	0	—
PC Memory Preindex	`([4,pc,d3:w:2],300)`	0	—	0	—
Total		**136055**	**100.0%**	**131014**	**100.0%**

Table 16 – 68020 Addressing Mode Use

The most notable difference in addressing mode usage was that the Gnu compiler is a little more clever in its use of register based address modes. It also seems to keep better track of addresses in registers, and consequently is able to use the register indirect, predecrement, and postincrement modes with greater facility.

However, of the 14 addressing modes available on the 68020, the three modes which are used the most frequently are (again) immediate, absolute, and register. Displacement mode and register indirect mode (the latter being a special case of the former, with a zero displacement) fill in the remainder of the main usage of the addressing modes. The more complicated modes involving indexing are used rarely or not at all. While we can see their occasional utility, we feel that the architecture would be better off without them. Again the question arises, "if simple instructions and address modes can perform the same functions as complex ones, and if the very complex functions of CISC architectures are rarely used, why are they present, instead of allowing the compiler to generate a sequence of simple instructions?"

5. Comparing Object Code Size

What we have seen so far is that CISC instructions and addressing modes are used inefficiently, and that RISC architectures are used much more efficiently. The question then arises "if RISC architectures require multiple instructions to execute what is done in a single CISC instruction, won't the program size on a RISC architecture be concomitantly larger?" The answer is quite surprising. Comparing the total instructions in Tables 1, 4, 7, 10, and 14, we see that the number of instructions on the RISC architectures is generally much larger than the CISC architectures. This might lead us to believe that the RISC architectures are less efficient in storing programs. However, the *number* of instructions is not the true measure of program size. Rather, it is the number of *bytes of memory* that the instruction occupy that is of concern.

Each instruction on the R2000, SPARC and MC88100 occupies 4 bytes of memory, and generally executes in a single machine cycle.† Although each instruction on the VAX typically occupies a single byte in memory (the `g` and `h` format floating point instructions take two bytes each), the operands for the instructions may require many bytes each. Each operand requires at least one extra byte of memory, and some complex addressing modes on the VAX require an extra 6 bytes to store. The execution speed on a VAX depends on the instruction, but generally the more complex an addressing mode, the longer the execution time. On the 68020, each instruction occupies two bytes of memory. Depending on the instruction

† Some instructions, such as the multiply or divide instructions of the R2000, always take multiple machine cycles. On the 88100 and the SPARC, instructions can take multiple cycles when the scoreboard register indicates a need to "stall" the pipeline. These instructions are generally the exception, and not the rule.

and the addressing mode, between zero and four extra bytes of memory are needed per operand. When we compare program size in bytes (as in the following table), we see some surprising results.

Architecture	Compiler	Number of Instructions	Size in Bytes	Fraction of VAX *pcc*	Bytes per Instruction
R2000	MIPS	118314	473256	1.59	4.00
MC88100	Green Hills	106192	424768	1.43	4.00
	Gnu	80834	323336	n/a†	4.00
SPARC	Sun	98694	394776	1.33	4.00
	Gnu	68748	274992	n/a†	4.00
VAX	Berkeley	67104	297928	—	4.43
	Tartan Labs	69377	313384	1.05	4.51
	DEC (VMS)	77769	280876	0.94	3.61
MC68020	Sun	89335	337024	1.13	3.77
	Gnu	84751	248828	0.83	2.78

Table 17 – Program Size Differences

As can be seen, the difference in program size is not so pronounced when we measure the actual number of bytes that the program occupies in memory. To be sure, the 30–60% increase in program size on the RISC architectures is of some concern. However, there are a number of issues to consider other than just instruction size.

1) The size of the instructions does not account for the total size of the program – statically and dynamically allocated data must also be considered. By examining the contents of the directories */bin*, */usr/bin*, */usr/ucb*, and */etc* on a μVAX, it was found that the number of bytes of instructions (i.e., *text* size) accounted for only 39% of the total number of bytes (i.e., *text+data+bss*) in the program. The percentage increase in program size becomes even smaller when dynamically allocated data (e.g., that allocated with *malloc*) is counted in the total.

2) Some compilers (e.g., the MIPS R2000 compiler) perform routine inlining and interprocedural optimizations. These techniques can lead to larger executable images, which nonetheless execute much faster than those generated without this technique. All compilers used for this paper were run with the highest levels of optimization available, and should not be penalized for generating efficient code that happens to be larger than inefficient code.

3) Most Unix systems, especially the newer RISC architectures, have demand paged executable files, so an increase in program size is not necessarily reflected in a larger memory image.

4) In spite of the extra instructions that need to be executed, RISC processors execute programs much faster than comparably priced CISC processors (the MIPS M/500 executes integer applications 6-10 times faster than the μVAX).

5) Memory density is increasing and memory costs are decreasing at breakneck rates. People nowadays think nothing of a μVAX personal workstation with 8 Mbytes of memory, when only 10 years ago some of the most powerful timesharing mainframes had only 1 Mbyte of memory. Any extra cost incurred in memory with the new RISC processors is well worth the benefit of increased throughput that the faster RISC processors provide.

As an aside, it is also interesting to note that the DEC (VMS) compiler produces smaller code than either the Berkeley or Tartan Labs compiler, in spite of having a higher instruction count. This is accomplished through an efficient use of the smaller sized addressing modes of the VAX.

† Because the Gnu compilers for the SPARC and the 88100 were untested, and because they occasionally dumped core and were unable to process some files, the ratio of code sizes between these compilers and *pcc* are not reported in this table.

6. What do the Results Tell Us?

Tables of figures are interesting in their own right, but what this paper sought to do was to provide *insight*, and not just statistics. A number of things can be inferred from the tables. The first is that within each architecture (where multiple compilers were evaluated), the instruction coverage and address mode coverage of the different compilers did not vary substantially. This tells us that although the different compilers were written by completely different groups, who at times had completely different goals (e.g., speed *vs.* space optimization), the overall functionality of the instruction set architecture was utilized in the same way by each group.

The addressing mode usage was nearly identical for every compiler within an architecture. Although they were not called out explicitly in any of the tables presented here, the individual instruction coverage (and not just the instruction class) was also very nearly the same within an architecture. From this statistic we may infer that if a feature of the instruction set architecture was valuable (i.e., a particular instruction or addressing mode), it was used. Conversely, if it was not valuable, it was not used.

The second point worthy of note is an observation that crosses architectural boundaries. Although the actual instructions differ between architectures, the use of the different *classes* of instructions (i.e., move, compute, and control) is also very similar.† This indicates that the test suite that was selected was a valid one – that although the programs and hardware varied, the compiler's use of the hardware for the programs was consistent.

Finally, when we examine the difference between the RISC and CISC architectures in this paper, we observe three things.

1) The reduced instruction sets are as adequate to the task of implementing (in assembly language) the test suite as are complex instruction sets,
2) The size of the resultant programs on the RISC architectures is not substantially larger than that on the CISC architectures, and
3) Generally speaking, this study showed that roughly 70% of RISC instructions are used by the compilers and 30% are not. For CISC compilers, the statistic is reversed – roughly 30% of the instructions are used, and 70% are not.
4) The "extra" instructions available on the CISC architectures are simply not used, and provide frills that cannot be taken advantage of by the compilers considered in this evaluation.

To be fair, we must point out that the tests only evaluated C compilers, and did not examine FORTRAN, Pascal, or Ada. It is entirely possible that compilers for these languages might generate slightly different code (especially in the case of Ada and Pascal, which would almost certainly use the "bounds check" instructions – the `chk` and `chk2` instructions of the 68020 and the `tbnd` instruction of the 88100 – to test the validity of array indices). However it is also the case that many compiler writers (Berkeley, MIPS, and Tartan Labs included) use the same (or substantially the same) code generator for different language front ends. In these cases, we feel that it would be unlikely that different language compilers would produce substantially different results than those we have seen here.

7. Conclusions

In all the machines examined in this report, the more complicated the instruction set architecture, the less utilized were the features of that architecture. While attractive to assembly language programmers, the complex features of CISC architectures are simply not used by compilers. Since the great majority of programmers write in high level languages (and not in assembly language), we feel that new architectures should be kept simple, to allow compilers to make full use of their features.

It has been amply shown in the past few years that RISC architectures execute faster than comparably priced and comparably sized CISC architectures. With the advent of GaAs technology, this speed

† The slight deviations (i.e., between VAX and other architectures) can be attributed to different number of registers (requiring more data motion where fewer registers are available), and to the difficulty of classifying an instruction such as "subtract one and branch if less than zero". We classified it as a conditional branch instruction, but it also could be considered a compute or comparison instruction.

differential will become even more pronounced. Since GaAs technology is currently limited by the size of the chips that can be produced, RISC becomes even more attractive, as it requires a smaller "footprint" to implement.

The term *reduced* has in no way implied *restricted*, nor has it caused the major increases in code size that CISC proponents claim will occur to support their cause. The slight increase in program size on the RISC processors is more than offset by a substantially faster execution speed. If Unix hopes to be the system of tomorrow in addition to that of today, manufacturers of Unix systems should concentrate their efforts more on RISC machines, than on CISC machines.

8. Acknowledgments

I would like to gratefully acknowledge the invaluable assistance of Tony Birnseth and Mike O'Dell for gathering statistics on compilers to which I did not have direct access, and to Robert Firth for his usual scathing (and technically brilliant) suggestions and commentary.

9. References

[1] Mario Barbacci, William Burr, Samuel Fuller, and Daniel Siewiorek, *Evaluation of Alternative Computer Architectures*, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, February 1978, Technical Report no. CMU-CS-77-EACA

[2] C. Gordon Bell, "RISC: Back to the future?," *Datamation*, 32(11), June 1986

[3] Mark Himelstein, *et. al.*, "Cross Module Optimization: Its Implementations and Benefits," In *Usenix Conference Proceedings*, June, 1987

[4] Daniel Klein and Robert Firth, *Final Evaluation of MIPS M/500*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1987, Technical Report no. CMU/SEI-87-TR-25

[5] Veljko Milutinovic, *et. al.*, "Architecture/Compiler Synergism in GaAs Computer Systems," *IEEE Computer*, 20(5):72-93, May 1987

[6] Marvin Minsky, *The Society of Mind*, Simon and Schuster, NY, 1986, p.127

[7] David Patterson, "Reduced Instruction Set Computers," *Communications ACM*, 28(1):8-21, January 1985

[8] Daniel Sieworick, C. Gordon Bell, and Alan Newell *(eds.)*, *Computer Structures: Principles and Examples*, McGraw Hill, New York, NY, 1982

[9] *Unix Assembler Reference Manual*, AT&T Bell Laboratories, Holmdel, NJ, 1979

[10] *VAX Architecture Handbook*, Digital Equipment Corporation, Maynard, MA, 1981

[11] *MC68020 32-bit Microprocessor User's Manual, Second Edition*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985

[12] *Assembly Language Programmer's Reference Guide*, MIPS Computer Systems, Inc., Sunnyvale, CA, 1986

[13] *The SPARC Architecture Manual*, Sun Microsystems, Inc., Mountain View, CA, 1987

[14] *MC8810 User's Manual*, Motorola Microprocessor Group, Austin, TX, 1988

Discuss: An Electronic Conferencing System for a Distributed Computing Environment

Ken Raeburn
Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
raeburn@athena.mit.edu

Jon Rochlis
Telecommunications Systems
Massachusetts Institute of Technology
Cambridge, MA 02139
jon@bitsy.mit.edu

William Sommerfeld
Apollo Computer, Inc.
Chelmsford, MA 01824
wesommer@athena.mit.edu

Stan Zanarotti
Laboratory for Computer Science[1]
Massachusetts Institute of Technology
Cambridge, MA 02139
srz@lcs.mit.edu

Abstract

As computers and computer networks become commonplace, electronic communication is rising in importance and utility. The challenge is to take a large, distributed computing environment and build a system which allows its users to communicate effectively and efficiently with each other. This paper compares and contrasts several common types of electronic communication, focusing on electronic conferencing. We describe the implementation of such a system, *Discuss*, for the computing environment found at MIT. Issues covered include the basic model of an electronic meeting, the currently implemented user interfaces, separation of user interface from underlying operations, splitting the workload between client and server, communications issues created by the heterogeneous environment, authentication and authorization, notification, and unification of the numerous subordinate UNIX libraries into a coherent whole. The paper next summarizes the *Discuss* system's current usage, then closes with a discussion of its possible future development.

[1]This project was supported in part by the Defense Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under contract number N00014-83-K-012.

Introduction

One of the most useful purposes computers and computer networks serve is helping people communicate. More and more experience is being gained in this area of computer use. While such communication has some problems (and certainly differences) when compared with traditional oral and written communication, it is not premature to claim success. At the same time, it is important to look at different types of electronic communication and the needs they meet (or fail to meet) with an eye toward possible improvements.

This paper presents the design choices made and implementation lessons learned during the development and initial deployment of *Discuss*, a computer conferencing system developed by MIT's Student Information Processing Board (SIPB)[2]. *Discuss* was designed for a heterogeneous distributed computer environment, such as that of MIT's Project Athena, and we pay particular attention to those issues which are important in such an environment.

Discuss has been in operation since 1986 and is currently used by more than 220 people at MIT. More than 100 *Discuss meetings* exist on eight server machines. Users and meetings exist in two separate administrative domains, Project Athena and the Laboratory for Computer Science (LCS).

This paper is organized as follows. We begin in Section 1 by describing the motivation for the *Discuss* project. This includes a description of the various types of electronic communication, an explanation of how conferencing differs from electronic mail and bulletin boards, and an overview of the MIT computing environment for which our system was developed. In Section 2 we define the model of a *meeting* which is the foundation of *Discuss*. Section 3 describes the currently implemented interfaces which allow a user to access meetings. The client/server model is explored in Section 4, and the communication issues caused by this split in duties are covered in Section 5. Following that, Section 6 describes the topic of authentication and authorization, showing how *Discuss* uses *Kerberos*[17] to provide authentication, but develops its own application level authorization scheme which we believe is required by a distributed conferencing system. User notification of new transactions via the *Zephyr*[4] notification system is covered in Section 7. We talk in Section 8 about the lessons learned, and conclude with some discussion of the future of *Discuss* in Section 9.

1 Motivation

Excluding telephones, fax machines, real-time video, voice conferencing systems and the like, the most common forms of electronic communication are electronic mail, interactive messages, bulletin boards, and conferencing systems. All are useful for decreasing the cost of face-to-face (or voice-to-voice) communication, but each has different strengths. In this section we will compare these forms of electronic communication, focusing on the advantages of conferencing. We will also describe the distributed computing environment at MIT into which any conferencing system must fit.

1.1 Types of Electronic Communication

Electronic mail is the most widely known and accepted method of electronic communication. Using the well understood metaphor of the postal service, even novice users can pick up the basic concepts without much effort. E-mail (as it is known) is generally batch oriented, and as in the postal system mail flows from a user to one or more recipients. Each user has some mechanism for retrieving his new mail (checking his mailbox) and for posting his mail for later delivery (dropping it in the outgoing mailbox). Mailing lists exist and facilitate communication with large numbers of people interested in the same topic. Each member of the list receives a separate copy of every message sent to that list. Due to variations in queuing delays, replies sent to recipients on different network nodes may sometimes arrive several days before or after the original message, causing much confusion.

Interactive messages have proven quite useful in a different way. Unlike e-mail, they are used only when the sender and recipient are both logged in. Instead of being queued for delivery whenever the mail system gets around to it, interactive mesages are usually delivered at once and displayed

[2] The SIPB is a volunteer organization dedicated to improving the student computing environment at MIT.

immediately on the receiver's screen. Interactive messages are useful for short, real-time queries ("When do you want to leave for the movie?" "Do you know where the source for discuss is?").

Electronic bulletin boards (or "bboards") allow users to "post" messages in a place which other users may peruse at a later date. Just as with real-world bulletin boards, bboards generally are restricted to specific topics and have large numbers of readers, frequently in the hundreds for dial-in microcomputer based systems, or thousands for the UUCP-based *netnews*[10]. The readers of an electronic bulletin board may be scattered throughout a state, country, or even the world, since, once posted, a message may be transmitted to other systems which store local copies for their users. These systems may also choose to forward the message to yet another computer. Messages generally expire after a few days or weeks, depending upon the disk space resources of the system or systems involved.

Electronic conferencing is subtly different from bboards; it is designed for a smaller number of users, frequently with privacy concerns, and with long term archival requirements. Not only do conferencing systems allow users to communicate with others interested in particular topics, but they prevent unauthorized users from reading such communication. For example, the professors and teaching assistants running a course might want to discuss possible quiz questions, but be confident that students would not be able to read their conversations (at least until after the quiz!). Also, there is frequently a requirement to retain the discussion for a long period of time so that people joining the organization at a later date may come up to speed by reading "what has gone before." While conferencing systems may have bboard-like facilities to periodically purge older entries, they are easily used for archival purposes and have more advanced mechanisms for sorting through previous entries than are usually found in bboard or e-mail systems.

In contrast to e-mail, the entries in a conferencing system are stored in one place so that the user sending a message need not know all the people interested in his message. Those interested will go to the message, rather than waiting for the message to come to them. Most importantly, this fulfills the goal of providing easily accessible archives; people can later decide they are interested in the message and access it as they would any other message in the conferencing system, rather than having to hunt around for possibly non-existent saved mail archives.

Since new messages in a conferencing system are immediately and globally visible, such a system can actually support running conversations with many participants. This is quite useful for problem-resolution meetings, in which many people may be able to fix a broken piece of equipment or answer a user's question in real-time. A dialog may be required and an archival record of the converation is needed to ensure proper resolution or simply to bring late-arriving helpers up to date on the current status of the problem.

Conferencing systems also help users avoid clutter in their personal electronic mailboxes. This is a natural result of conferences being based on specific topics. Of course, vague or overly general meeting topics may reduce this gain, underscoring the importance of the role of a moderator, or *chairman*, to ensure that conversations stay on track, perhaps by suggesting that further discussion take place in a different meeting or by creating a new meeting if one with suitable topic does not already exist.

1.2 The Distributed Services Environment

MIT's Project Athena[1][20] is an example of a computer utility based on the distributed services model of computation. Athena has approximately 850 4.3BSD UNIX workstations (of several different hardware types) scattered around campus, in locations ranging from public terminal rooms to private offices to living groups. Workstations and servers are connected by a high speed network. Each workstation has a moderate-sized hard disk used for storing essential software and for swap space. Most system software and user files are obtained from remote file servers. The location of both the system software and a user's filesystem are determined by a network name service called *Hesiod*[6]. The network authentication service, *Kerberos*, is essential because the network is unsecured and a public workstation is under total control of its current user. (The root password is even published![3])

[3] For the curious, the root password is "mrroot."

```
[0001]*  (2) 10/29/87 12:43 raeburn          Reason for this meeting
[0002]  (33) 10/29/87 17:06 raeburn          error handling
[0003]   (2) 10/29/87 17:22 srz@LCS.MIT.EDU  Re: error handling
[0005]  (16) 10/30/87 11:06 srz@LCS.MIT.EDU  send_to_kdc and use of resolver
[0006]  (20) 10/30/87 11:41 jon              Re: error handling
[0007]   (5) 11/01/87 23:10 jis              Re: send_to_kdc and use of resolver
```

Figure 1: Sample listing of transactions in a *Discuss* meeting

Electronic mail service is provided by a central mailhub, which is responsible for re-writing the headers in order to present a unified address-space and for distributing messages for internal users to a post office server. Users can retrieve their mail from their post office server, located using *Hesiod*, via the Post Office Protocol modified to use *Kerberos* authentication. The *Zephyr* notification system provides delivery of interactive messages from users to other users or from service-providing processes to users requesting notification. *Zephyr* also makes use of *Hesiod* to find its servers and *Kerberos* to provide authentication of messages.

1.3 Enter *Discuss*

The SIPB undertook the task of developing an electronic conferencing system for this environment in 1986. Much of *Discuss* came out of the SIPB's experience with the Multics conferencing system *forum*[9] and a desire to see it extended beyond a single machine environment[13]. The part-time efforts of Ken Raeburn, and Bill Sommerfeld, and Stan Zanarotti have resulted in a solid production-level system, which has been adopted for use by Athena's System Development and User Services Groups, and portions of MIT's Laboratory for Computer Science (LCS). SIPB, Athena, and LCS run *Discuss* as a network service for themselves and the MIT community.

2 *Discuss* Meetings

The *meeting model* is the central concept in *Discuss*, as is the case with Multics *forum*. Users "attend" a set of electronic meetings, just as you might attend a regular set of face-to-face meetings. For each meeting attended you listen to what other participants have to say, and you can contribute as you wish. However, unlike face to face meetings, not all the participants in a *Discuss* meeting need be present at the same time. Participants need not be geographically close, so long as there is a network connection between the user and the machine on which the meeting is stored. Removing the time and space constraints present in face-to-face meetings is one of the major advantages of electronic conferencing.

A meeting is a set of *transactions*. Each transaction is labeled with the author (unless the author wishes to remain anonymous), a subject, the time of entry, and a unique transaction number. The contents of a transaction are stored as uninterpreted eight bit data, though the existing client user interfaces are only prepared to receive ASCII text. *Discuss* keeps track of which transactions a user has seen, and has the ability to only display transactions which are new to the user.

Transactions about the same subject are grouped in subconversations called *chains*, and tools are provided for moving through such chains. When a user replies to a given transaction, the subject of the reply is created by prepending "Re:" to the subject of the original transaction; but, more importantly, a pointer is associated with the new transaction which indicates which transaction is the original. The original transaction also has a forward pointer to the reply. In this manner a chain of transactions can be created and easily walked through. If one replies to a transaction in the middle of a chain, *discuss* simplifies the pointers by pretending that you have replied to the transaction at the end of the chain. Each transaction in a chain has a previous reference (`pref`) and a next reference (`nref`), except for the first (`fref`) and last (`lref`) transactions in the chain, which only have an `nref` or `pref`, respectively.

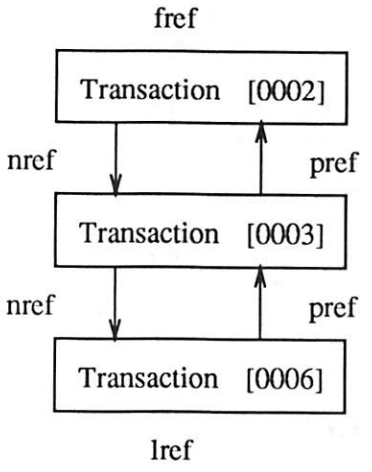


Figure 2: Transactions form chains.

Transaction numbers are assigned in a strict order, unlike the *notes*[7] conferencing system, and unlike the *netnews* bboard system they are consistent for all readers of the meeting. Replies are simply given the next transaction number and threaded to produce a chain as described above. If transactions are deleted, the transaction number namespace is left unchanged. Users can refer to a transaction by number with the confidence that the transaction number will not change in the future.

Figure 1 shows an example listing of transactions. Transaction 4 has been deleted. There are two chains, one starting at transaction 2, and the other at transaction 5. The "error handling" chain has an `fref` of transaction 2 and an `lref` of transaction 6. All the transactions in the chain have `pref` and `nref` pointers as shown in Figure 2.

Entering a message in more than one meeting while maintaining the ability of the user to only see the message once is not supported. Unlike other conferencing systems, like Multics *forum*, you cannot tell who attends a meeting (unless they speak up, of course) nor who has read a given transaction.

3 User Interface

The original interface was selected for its simplicity and, for some of us, familiarity. It is a terminal-oriented, shell-like interface which loops through a prompt-input-execute sequence.

The user is presented with the idea of a meeting being attended, and a "current" transaction within that meeting; he may go to a different meeting with the `goto` command, or run one of many commands within a meeting. The commands available for examining the contents of a meeting include `list` and `print`; each command takes one or more transaction numbers as arguments. Transaction numbers may be specified as integers or ranges of integers, or symbolically, such as `current`, `pref`, or `first`.

Naturally, some method for entering transactions is needed. We provide the `talk` command for entering transactions to start new chains, or to stand by themselves, and `reply` to enter a response to an existing transaction. The `talk` command requests a subject, then allows the user to enter the text of his message. It uses a simple line-input interface, terminating the message on input of a line containing only a '.' or escaping to run an editor program specified by the environment variable "EDITOR" on `~e`. Although the protocol does permit new subjects for any transaction, the `reply` command currently assumes a subject derived from the message being replied to, prefixed with `Re:` if it is not already present. A `delete` command is also provided.

There are also commands for dealing with several meetings. Aside from the `goto` command mentioned above, we have provided the `check_meetings` command, which checks whether any meetings

```
get_mtg_info (mtg_name) returns (mtg_info, err_code)
get_trn_info (mtg_name, trn_num) returns (trn_info, err_code)
get_trn (mtg_name, trn_num) returns (trn_text, err_code)
add_trn (mtg_name, trn_text, subject, reply_trn_num)
        returns(new_trn_num, err_code)
```

Figure 3: Sample set of *Discuss* server operations

the user attends have changed since the user last attended them. The command `next_meeting` can be used to step through this list meeting by meeting, allowing the user to view new transactions in all meetings he attends.

4 Client/Server Model

One of the goals of *Discuss* is that it is distributed. Unlike its timesharing-based relatives, *Discuss* allows meetings to be shared among users working on different machines. The design of *Discuss* addresses the problems of distributing its functions across different machines.

The MIT computing environment, as described in Section 1.2, influenced the design of *Discuss*. Almost all MIT computers are connected to a set of high-speed networks and gateways, allowing fast data transfer between any two machines on campus. In addition, most students use public Athena workstations, which are single-user UNIX machines with graphic displays. These workstations have spare CPU time, but retain no local state. MIT computers form a loose federation of autonomous systems. In particular, MIT has no common, network-wide file system that could store all *Discuss* meetings. This differs from the Carnegie Mellon bboard system[2] which uses the Andrew File System to distribute meetings among many machines.

Given the MIT environment, *Discuss* uses the client/server model of computation. With this model, an application is divided into two parts. One part, the server, manages a resource, while another part, the client, accesses this resource across the network by contacting the server. The client/server model is a common model for other applications in the Athena environment; it encapsulates the particular resource within the servers.

4.1 The *Discuss* Server

With *Discuss*, the particular resource is the set of meetings on a given machine. A server provides access to these meetings, allowing clients to manipulate them in a controlled fashion. The pool of *Discuss* meetings is distributed among various servers; unlike a *netnews* group, a *Discuss* meeting is located on a single machine. This allows all participants to have a consistent view of the meeting, but requires low-latency network connectivity. The transactions are numbered consistently, and users can immediately see new transactions entered by other participants.

A *Discuss* server provides a low-level interface for manipulating meetings. The operations are fairly primitive, allowing clients to get information about single transactions and single meetings. The server simply manages the meeting data, ensuring its consistency and controlling access to it. The server provides a core set of operations that can be used by any user interface. A sample set of these primitives is shown in Figure 3.

On a UNIX *Discuss* server, a meeting is stored as a directory owned by the `discuss` user on the server machine. The name of the meeting is the name of this directory; `/usr/spool/discuss/ping` is a sample *Discuss* meeting name. The meeting can be located anywhere within the server machine's filesystem. This allows users with accounts on a server machine to create *Discuss* meetings under their home directories. A *Discuss* meeting contains three files: `transactions`, `control`, and `acl`. The `transactions` file contains the text of the transactions, stored in an append-only format. The `control` file contains indexes into the `transactions` file, as well as information about transaction chains and deleted transactions. The `acl` file contains the access control list for the meeting. Access control is described later in this paper.

Discuss meetings should be placed on machines that have sufficient amount of disk space to support the meetings. Since *Discuss* meetings are archival in nature, this disk space should be backed up regularly[4]. *Discuss* server machines should also have enough computing power to support the expected number of clients.

4.2 The *Discuss* Client

A user accesses *Discuss* meetings by running one of the various *Discuss* client programs. This program manages the user interface and contacts the appropriate servers to manipulate the meetings that the user attends. The client program displays the information contained in a meeting and allows the user to enter new transactions.

Since the client/server interface is low-level, it does not dictate a particular user interface; different clients can present different user interfaces, provided that they fit into the meeting model explained above. Currently, there are five different *Discuss* clients: `discuss`, `xdsc`, a `Common Ground` implementation[8], `dsmail` and an emacs[16]-based client. `discuss` is the general command interface described above. `xdsc` is an experimental client based on the X window system[15]. `dsmail` is a utility program that automatically enters mail messages into *Discuss* meetings, trimming the headers appropriately.

In the client/server interface, a *Discuss* meeting is named by its server machine name and its location on the server machine. Since these names are difficult for humans to use, the client program accepts simple mnemonic names for meetings, and converts them into this low-level form. This mapping could be provided by a name service such as *Hesiod*, but none existed at the time *Discuss* was initially designed. Instead, a user file called `.meetings`, in the user's home directory, is used. This ASCII text file records what meetings the user attends, storing its low-level name and the simple names that can be used to refer to the meeting. This allows users to have personal aliases for the meetings they attend. The `.meetings` file also contains information about what transactions the user has seen in a given meeting. This information is kept on the client side because of a desire to keep the server simple, and also because of concerns that such information shall be private.

5 Communications

To communicate between clients and servers through the network, *Discuss* defines a protocol that allows clients to perform different *Discuss* operations on servers. Like other systems, *Discuss* uses several layers of network protocols. These protocols are shown in Figure 4. We give a brief description

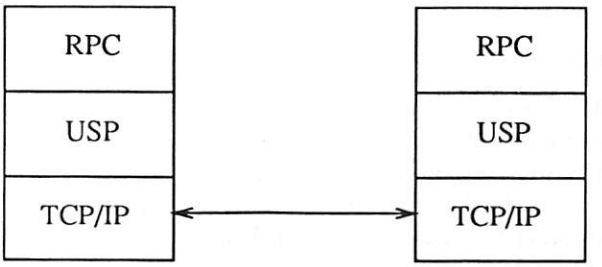


Figure 4: Protocol layering in *Discuss*

of each protocol and why it was chosen. We start with the highest-level protocol, and proceed down through the lower levels.

RPC A *Discuss* server provides a set of operations that clients can perform on meetings. These operations are presented within a Remote Procedure Call (RPC) mechanism. RPC is a well-understood

[4] A lesson learned the hard way.

programming paradigm which allows the client-server interface to look like a programming interface. The exact RPC mechanism that *Discuss* uses is home-brew, because the recent RPC mechanisms, such as Apollo NCS[5] and Sun RPC[19], were not available when *Discuss* was first built. One nice feature of the *Discuss* RPC mechanism is that it is designed so that the client and the server can be linked together into a single program. This feature is useful when debugging or writing utilities that need to run quickly.

USP The Remote Procedure Call mechanism uses Unified Stream Protocol (USP) [3] to transmit data values for the RPC. USP was an early protocol to transmit integers, strings, and other information in a machine-independent way. USP solves the problems of byte-ordering and how strings should be transmitted over a stream. These days, newer protocols, such as Sun XDR[18] or ISO ASN.1[11], would be used to transmit these values.

TCP/IP USP is built on top of TCP/IP, the transport layer of the Internet protocol suite. TCP/IP provides a high-speed, reliable stream between any two hosts connected on the Internet. TCP/IP is a popular protocol, with implementations for machines ranging from mainframes to microcomputers. This allows *Discuss* to be used on different operating systems, such as UNIX and Multics.

Although the current implementation of *Discuss* uses these protocols to communicate between clients and servers, *Discuss* could use any other protocol that provides the same functionality. *Discuss* clients and servers only deal with a set of meeting operations, which are available as ordinary procedure calls. *Discuss* could be implemented on other RPC mechanisms, such as Sun RPC, Apollo NCS, or the Mercury RPC[12], without much of a problem. Alternatively, the operations could be encoded in ASCII on a standard TCP stream, as SMTP[14] is.

6 Authentication and Authorization

Many existing networked conferencing systems are explicitly insecure—they make no guarantees about the integrity, confidentiality, or authenticity of messages. They also lack a flexible authorization system to control access to messages. Messages can easily be "forged" — appearing to come from a user who did not send them — or sent anonymously. There is rarely any effective control over who can enter a message in a conference, or any way to keep messages private. *Discuss* answers these issues by using an external authentication system (*Kerberos*) in combination with its own authorization mechanisms.

6.1 Authentication

In an open network system, a network service cannot trust the integrity of its clients; they can lie about the identity of their users. Through the use of an encryption-based third-party authentication system, such as *Kerberos*, it is possible to allow servers to verify the identities of their clients. Thus, a *Discuss* server can use *Kerberos* to be sure that a supposed author really did originate a given transaction, and can label the transaction with his authentication name.

If the meeting is located on the same machine as the client, *Discuss* does not use *Kerberos*; instead, it relies entirely on the security mechanisms inherent in UNIX. The *Discuss* library spawns a `setuid` back-end program to handle the requests. The back-end knows the real user ID of the front-end, and uses the associated user name as the authenticated name of its client. The mail-to-*Discuss* gateway uses this system; mail messages are entered from `daemon` (the username under which `sendmail` runs filter processes), with the original `From:` line of the mail message appearing in the body of the transaction.

The result of the authentication process, whether through *Kerberos* or through UNIX mechanisms, is an authenticated name which is used as the basis for authorization.

6.2 Authorization

Once it has an authenticated name, the *Discuss* server must then make authorization decisions. One of our early design goals was to provide fine-grained access controls; that is, it should be possible to control the access to a meeting down to the accuracy of a single user.

Each meeting has an access control list (ACL), listing pairs of authenticated names and the associated access rights they have to the meeting. There are seven distinct permissions:

answer allows the user to reply to an existing transaction,

chairman allows the user to change the access control list.

delete allows the user to delete any transaction in a meeting.

read allows the user to read any transaction in a meeting.

owner allows the user to read, delete, and retrieve transactions that he entered; usually, chairmen are willing to allow a user to "take back" what he said, but some might not want to allow that.

status allows the user to find out summary infomation (last modified time, number of transactions, etc.) about the meeting.

write allows the user to start a new chain.

It should be noted that no one permission directly implies another. For example, chairman access does not imply read access.

A user name of "*" in an access control list gives the rights for users who are not explicitly mentioned on the ACL; in addition, a user name of "???" matches unauthenticated users. A meeting chairman can prevent anonymous transactions by denying answer or write permission to unauthenticated users.

In addition to providing control over who can access a meeting, *Discuss* also provides control over creation of meetings; specifically, an access control list file named `acl` must exist in the directory in which the meeting is created, and that ACL must grant 'a' access to the user who is attempting to create the meeting.

In the great UNIX tradition, there is currently no quota system limiting the amount of disk space that an individual meeting can use. A system administrator may impose a disk quota for the `discuss` pseudo-user; the total will apply to all meetings. *Discuss* handles disk-full or over-quota conditions gracefully.

7 Notifications

Since a meeting has a single location, other users can immediately see any new transactions added to a meeting. This adds a real-time flavor to *Discuss*; users can reply to transactions immediately, and create an on-going conversation among several participants. To facilitate this, *Discuss* allows users to be notified when a new transaction is added to a meeting.

Discuss uses the *Zephyr* Notification Service to inform interested users of new transactions in meetings. *Zephyr* is a notice transport system that delivers interactive messages in a networked workstation environment. Users register which classes of messages they are interested in; *Zephyr* will deliver messages to the appropriate users.

When a *Discuss* server adds a new transaction to a meeting, it sends out a *Zephyr* notice. Since this notice contains information such as the subject and author of the transaction, the transmission of the notice should not violate the privacy of the meeting. The server does not send out public notifications of new transactions in private meetings. If the meeting is publically readable (read access to "*"), it sends out the *Zephyr* message to recipient "*." Anybody can subscribe to these messages. If the meeting is private, then the *Zephyr* message is only sent to those users who have read access to the meeting.

8 Client Tools

When writing the first *Discuss* client program, we came across a number of problems that had to be solved. Instead of solving these problems in a limited way, we attempted to build general tools that could find use in different situations. These tools have found their way into other *Discuss* clients, and other software projects.

This section describes the tools that were built to support *Discuss*. These tools are the *Discuss* library, which contains general routines that can be used by any *Discuss* client, the subsystem (*ss*) library, which supports command-line interfaces, and the Common Error-handling (*com_err*) package, which allows multiple packages to report errors in a coherent way.

8.1 The *Discuss* library

When we wrote our first *Discuss* client program, we were aware that there would be those who would not like the style of interface that we chose, and who would want some other interface to use. We wanted to be able to build clients with different interfaces without resorting to "sharing via text editor."

To meet this goal, we designed the *dsc* library. It performs all interaction with *Discuss* servers (via RPC calls) as well parsing and updating a user's `.meetings` file. Thus the user interface needs only know how to call these routines and need not know any of the implementation details.

Using this library, a prototype X-based client `xdsc` has been written. Although we found one or two very minor problems in the actual implementation, we believe the modularity imposed by the *dsc* library has greatly reduced the effort required to develop new user interfaces.

8.2 A Simple User Interface

Another package we wrote for this project was the original shell-like user interface. While the routines implementing the specific commands are very particular to *Discuss*, the driver that prompts for them is not. The package we use, known as *ss*, provides the input and dispatch routines for not only our own original *Discuss* client program, but also a control program used by *Zephyr*, one of the *Kerberos* administrative utilities, and various other programs which need a simple interactive shell-like interface. This section gives a brief overview of the use of this package.

This subsystem package consists of a command table translator and a run-time library. The command table translator converts a descriptive list of commands (including several command names, a short one-line description, and a subroutine name for each) into a C source file which is compiled and linked into the the application, along with the run-time library. The *ss* library consists primarily of a dispatch routine which prompts for input, reads and parses a command line, and calls one of the subroutines listed in the command table. A user of this system calls a routine to create a subsystem object, and then invokes a dispatch routine with that subsystem object as a parameter. The routines are called with "argc, argv" arguments similar to the C function `main`, along with a third parameter indicating which subsystem object the command was invoked from.

This subsystem package includes as built-in commands a "?" command, which lists the commands available (and their descriptions), and some simple commands like "quit". It manages input handling, primitive shell-like argument parsing (breaking up a string into an array of words), and a help utility that makes use of programmer-supplied directories of help files (to be viewed with "more").

Although this package lacks such features as intelligent argument handling and command and argument completion, it is extensible and still under development. Even without these features, we have found this subsystem management package to be useful for a number of applications.

8.3 Error handling

One of the aspects of the *Discuss* client programs is that they use different packages within the same program. These packages come from different sources, and each has its own way of reporting error conditions. Although UNIX has well-defined error handling for kernel routines, there is no standard

way for libraries to return errors. In many cases, each package defines a set of error codes, and provides a separate method of converting this error code into an error message. For some procedure calls, an error may arise from several sources. In this situation, there is no way for the calling procedure to understand the error.

In response to this problem, we created the *com_err* package. It allows multiple libraries to return error codes in a coherent manner. With *com_err*, errors are represented as 32-bit unsigned integers. These integers can be returned by functions, passed between different libraries, and sent over the network. The *com_err* library supplies a routine to convert these integers into error messages.

To use *com_err*, the programmer first creates a text file describing the error table. This text file names the error table, and lists the symbolic names of the error codes and the error messages associated with them. The *error table compiler* reads this text file, and produces two files: a C source file that defines the symbolic names as C-preprocessor symbols, and an object file containing the error messages. The programmer includes the C source file to obtain definitions for the error codes, and links the object file into the application.

To avoid collisions between error codes, *com_err* uses the error table name to generate the numbers for error codes. Error tables are named using four letters. *com_err* collapses these letters into the top 24 bits of the error code, leaving 8 bits to differentiate among different errors in the same error table. This restricts error tables to at most 256 entries, which has proven more than adequate. The first 256 error codes are reserved for UNIX errors, so that UNIX errors can be treated as *com_err* codes.

We added *com_err* to *Discuss* by generating error tables for the different packages it used. For packages that have their own error codes, such as *Kerberos*, the caller of the package converts the returned value into a *com_err* codes by adding a constant. All *Discuss* routines return standard *com_err* error codes.

com_err provides a flexible approach of handling error codes, and has been adopted for other projects. For example, *Zephyr* uses *com_err* to encode and transmit error indications. *com_err* provides the glue that allows multiple packages to be integrated smoothly.

9 Status and Future Directions

Discuss has found ever-increasing popularity since its initial implementation in Fall 1986. Its user community has grown from a core set of SIPB members to more than 220 MIT students and staff. The number of meetings has grown from a few test meetings on a couple machines to 116 meetings on eight server machines.

9.1 Current *Discuss* meetings

Some of the current *Discuss* meetings archive mailing lists that exist on the Internet. For example, there are *Discuss* meetings that archive the "Info-Mac", "TeXHaX", "Risks", and "Kerberos" mailing lists. These meetings allow MIT computer users to read these lists without getting themselves added to the lists and having these messages intermixed with their regular electronic mail.

Discuss meetings support several services at Project Athena. They archive user bug reports, so that developers can review them in a readily accessible and efficient manner. *Discuss* is also used by the Athena On-line Consulting (OLC) System to store sessions between users and consultants. This way, all consultants can review interactions between users and consultants.

Several Athena development projects use *Discuss* to coordinate the activities of project members who have different schedules and different geographic locations. *Discuss* has been used to help develop *Discuss* itself, *Zephyr*, and *Kerberos*. With a *Discuss* meeting, chaining allows replies to be organized, and the archival nature of *Discuss* means that new members of the project can review past transactions.

Finally, *Discuss* meetings play a social role at MIT. There are *Discuss* meetings for random discussion, changed *plan* files, depressing thoughts and cheery thoughts, as well as private meetings for self-selected groups of people. These meetings frequently generate interesting, lively discussion.

9.2 Future of *Discuss*

At the moment, *Discuss* appears to have attained critical mass, in that the current user community is producing enough interesting information to sustain itself. Future development in *Discuss* will be aimed at improving the current system, and increasing its usage within the MIT community.

The next improvement to *Discuss* will be better user interfaces, so that *Discuss* can be used by people who are unfamiliar or uncomfortable with command-line interfaces. Currently, one of the authors is working on `xdsc`, an X-based client, so that users can read and enter *Discuss* transactions using a mouse/window interface. Minor changes such as displaying a user's real name instead of his user name could have a large user friendliness payoff. Another way of increasing *Discuss* usage is to port *Discuss* onto other machines with distinctive user interfaces, such as the Apple Macintosh.

Along with different interfaces, we are considering adding new features to *Discuss*. With an X-based interface, it is useful to consider multi-media transactions, where *Discuss* transactions could mix pictures and text. *Discuss* could also be better integrated with mail, so that users could reply to *Discuss* mail archives directly. Another natural extension is the idea of voting, so polling can be conducted in an authenticated and convenient manner. *Notes* and *Forum* have shown that a "chairman message", shown to all users of a meeting or even just users about to enter a new transaction is useful; adding such a feature to *Discuss* would not be difficult. Per-transaction or per-chain "attributes" have been requested for meetings which track tasks, so tasks represented by transactions or chains could be marked as "resolved" or "urgent". This would be more difficult to do in a general way, but would probably be worth the effort.

Integrating a nameservice like *Hesiod* into the meeting location process would simplify finding new meetings and would allow meetings to be easily moved from one server to another. At the moment this information is scattered among all the `.meetings` files. Since we are not sure that the benefits from this are worth the cost of centralized registration of meetings, this is still a topic for further thought.

Another open issue is that of server-side state. Currently the *Discuss* server retains no per-user state. This greatly simplifies its implementation, but it makes the `.meetings` file more important and does not allow one to discover who else is attending a meeting or which transactions they have read. Some people feel having this information available would be useful, others think privacy concerns preclude releasing such information.

10 Acknowledgments

We would like to thank Robert French, Ted Ts'o, and Brian LaMacchia for their work on the implementation of *Discuss*; Nancy Gilman for her efforts to document *Discuss* for mortal users; Brian LaMacchia, Andrew Berlin, Dan Geer, Judith Provost, Ron Hoffmann and others for reviewing early drafts of this this paper; the fellow members of the SIPB[5] for fostering the kind of supportive environment that leads to work such as *Discuss*; and most of all, the users, who have made *Discuss* a success.

References

[1] E. Balkovich, S. R. Lerman, and R. P. Parmelle. Computing in Higher Education: The Athena Experience. *Communications of the ACM*, 28(11):1214–1224, November 1985.

[2] Nathaniel Borentstein et al. A Multi-media Message System for Andrew. In *USENIX Association Winter Conference 1988 Proceedings*, pages 37–42, February 1988.

[3] David D. Clark. Unified Stream Protocol. Technical Report CSR RFC 272, M.I.T. Laboratory for Computer Science, February 1985. Internal Working Paper.

[5] All of the authors are members of the SIPB. Ken, Jon, and Stan have served as Chairman of the SIPB.

[4] C. Anthony DellaFera et al. The *Zephyr* Notification Service. In *USENIX Association Winter Conference 1988 Proceedings*, pages 213–220, February 1988.

[5] Terence H. Dineen et al. The network computing architecture and system: an environment for developing distributed applications. In *USENIX Association Summer Conference 1987 Proceedings*, 1987.

[6] Stephen P. Dyer. The *Hesiod* Name Server. In *USENIX Association Winter Conference 1988 Proceedings*, pages 183–190, February 1988.

[7] Raymond B. Essick IV and Rob Kolstad. Notesfile Reference Manual. *UNIX User's Supplementary Documents*, February 1983.

[8] Chris Hancock. Common Ground. *Byte*, 10(13):239–246, December 1985.

[9] Honeywell Information Systems. *Multics Forum Interactive Meeting System User's Guide*, CY74-00 edition, July 1982.

[10] Mark R. Horton. How to Read the Network News. *UNIX User's Supplementary Documents*, April 1986.

[11] Information processing systems — Open Systems Interconnection. Specification of Abstract Syntax Notation One (ASN.1). Technical Report ISO 8824, International Organization for Standardization., December 1987.

[12] B. Liskov et al. Communication in the Mercury System. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, pages 178–187, January 1988.

[13] Jay Pattin. A Design for a Networked Computer Conferencing System. Technical report, Massachusetts Institute of Technology, May 1983. Bachelor's Thesis.

[14] J. B. Postel. Simple Mail Transfer Protocol. Technical Report NIC/RFC 821, Network Working Group, USC ISI, August 1982.

[15] R. W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1987.

[16] Richard Stallman. *GNU Emacs Manual*, sixth edition, March 1987.

[17] Jennifer Steiner, Clifford Neuman, and Jeffrey Schiller. Kerberos: An Authentication Service for Open Network Systems. In *USENIX Association Winter Conference 1988 Proceedings*, pages 191–202, February 1988.

[18] Sun Microsystems, Inc. XDR: External Data Representation Standard. Technical Report NIC/RFC 1014, Network Working Group, USC ISI, June 1987.

[19] Sun Microsystems, Inc. RPC: Remote Procedure Call Protocol Specification Version 2. Technical Report NIC/RFC 1057, Network Working Group, USC ISI, June 1988.

[20] G. Winfield Treese. Berkeley UNIX on 1000 Workstations: Athena Changes to 4.3BSD. In *USENIX Association Winter Conference 1988 Proceedings*, pages 175–182, February 1988.

A Partial Tour Through the UNIX† Shell

Geoff Collyer

Department of Statistics
University of Toronto
Toronto, Ontario, Canada
M5S 1A1
utzoo!utstat!geoff
geoff@utstat.toronto.edu

ABSTRACT

We have recently completed protracted surgery on the UNIX command interpreter or 'shell' [Bourne1978] to make it use the standard UNIX memory allocator (*malloc* (3) and relatives) for its internal memory management instead of the original scheme (catching its own memory faults, using the *sbrk* (2) system call to grow its memory allocation and restarting faulting instructions). We also fixed some bugs, *lint* (1) complaints and suboptimal performance. This paper describes the lessons learned about the internal workings of the shell. Much of this information is oral folklore or is simply not generally known, and requires a determined effort to learn, yet is essential to correct understanding and maintenance of the shell.

1. Introduction

A very sketchy overview of the shell is that it parses its input into a parse tree, then walks the tree, executing the tree nodes by creating pipes, *fork* ing, redirecting I/O descriptors, *exec* ing commands, and the like. Interwoven with this are macro expansion; honouring quoting; coping with keyboard, alarm clock, and other interrupts (via signals); and maintaining variables and functions. One can think of the shell as a macro processor which also interprets commands.

The original Seventh Edition shell had to run in a small address space (64k bytes of instructions and 64k bytes of data, including stack segment), yet places no arbitrary limits on lengths of strings or input lines (which may explain some of the contortions in the code). With the exceptions of *eval* and quoting, which are incompletely specified, the external specification of the shell is simple, rational, and clean. No comments in this paper should be taken as denigrating these achievements.

The shell source code is opaque and under-commented in spots, which causes maintainers to attempt only minimal changes and fixes. The shell was the last program ported to the Interdata during the original UNIX port, [Johnson1978] due to the difficulty of getting the details of restarting faulting instructions just right, which is why the Seventh Edition (also known as "V7") distribution tape includes `/bin/osh`, the Sixth Edition shell. [Ritchie1987] One cliched complaint about the original shell source, that it was written in a dialect of the C language resembling Algol 68, is not a problem once one

† UNIX is a trademark of Bell Laboratories.

gets used to it, particularly if one has a reading knowledge of Algol 68. In any case, recent System V shells are written in ordinary C. However, this is the least of the problems.

Upon attempting to make the Ninth Edition shell (derived from the System V Release 2 shell) run on a Sun-3 under SunOS 3.x, we ran into trouble with the shell's peculiar internal memory management. The shell often failed in a spectacular and annoying way: it grew its stack segment to maximum size and then dumped core. We discovered many previously-undocumented characteristics of the shell in the process of converting the shell to use *malloc* (3) and relatives. The result of this work is referred to throughout this paper as ''the new shell'', for lack of a better name. This paper discusses only the parts of the shell visited in the course of making it work correctly on the Sun-3; the rest of the shell is relatively straightforward.

The rest of the paper consists of six sections: section 2 describes the design of the old shell, section 3 the problems in implementation of the old shell, section 4 the fixes we applied to produce the new shell, section 5 our methods, section 6 our conclusions, and section 7 acknowledgements. Those readers interested only in the internals of the old shell may safely skip sections 4 and 5.

2. Design

2.1. Memory Management

The first subtlety encountered by most shell maintainers is the shell's internal memory management, which has been characterised as 'extremely elegant, but a house of cards'. The shell contains its own memory allocator, a variant of the Seventh Edition *malloc*, which maintains two distinct kinds of storage: *heap* storage, which has an indefinite lifetime and resembles ordinary *malloc*ed memory; and *''stak''* storage (so spelled to distinguish it from the shell's stack segment) which is allocated and deallocated in strict last-in, first-out order. Heap and *stak* storage blocks are intermingled in the data segment.

A fundamental abstraction of the shell is a stack of *stak* storage, in which the top item on the stack is typically a growing string. The top item may be moved at the convenience of the memory allocator, so it should (in theory, if not always in practice) only be referred to by the functions and macros of `stak.h`; keeping private pointers into the top item is forbidden. Once the top item has been grown to its maximum extent, it may be made permanent and immovable, and a new, empty top item is begun.

The interface to the *stak* storage manipulators, `stak.h`, declares several functions and macros, notably *pushstak(byte)*, which appends a *byte* to the top item and advances the top past it, acquiring more memory from UNIX as needed. *relstak()* yields the integer offset of the top of the top *stak* item, i.e. the size of the top item. *absstak(offset)* yields a temporary pointer to the top item, *offset* bytes in; this pointer must not be retained. *setstak(offset)* sets the top of the top item to be *offset* bytes in. *zerostak()* stores a zero on the top of the top item but does not move the top. *curstak()* yields the a temporary pointer to the top of the top item; this pointer must not be retained. *usestak()* calls *locstak()* and ignores the result. *fixstak()* calls *endstak* with a pointer to the top of the top item.

locstak() returns a temporary pointer of the bottom of the new top *stak* item, which will be big enough for any structure used in the shell (notably `struct fileblk` or `2*CPYSIZ` from `io.c`); this pointer may be used until one of *pushstak*, *endstak*, or *fixstak* is called. *endstak(argp)* terminates the top item at *argp* with a zero byte, makes the top item permanent, starts a new top item, and returns the address of the terminated and now permanent item. *getstak(n)* returns a permanent item of size *n* bytes by growing the current top item. *savstak()* asserts that the top item is empty and returns the address of the bottom of the top item. *tdystak(ptr)* removes temporary files (e.g. from here documents) described by structures on the *stak* down to address *ptr*, and pops the *stak* down to but not including *ptr*. *stakchk()* reduces the data break if possible. *cpystak(string)* copies the *string* to a new permanent item and returns its address.

The memory allocator and other parts of the old shell assume that the address of newly-allocated *stak* storage will be greater than the addresses of all other still-active *stak* storage. (This is not true of storage obtained from arbitrary *malloc* s.) This property is exploited by tricks such as recording a single ''watermark'' pointer, to mark the points in several intertwined stacks of *stak* storage above which data may be eventually discarded, then later popping the top items from the stacks by popping each stack until its stack pointer reaches the watermark pointer, or drops below it.

Heap storage is allocated by *alloc* (also known as *malloc* in the old shell). The shell assumes fundamentally that *free* will ignore attempts to free the address zero (''null pointers''), addresses in the shell's stack segment (automatic variables, command-line arguments, and environment variables), and addresses of *stak* storage not yet made permanent and immobile; the shell's *free* is meant to free only heap storage and permanent *stak* storage.

The old shell catches its own memory faults (via the SIGSEGV signal, typically caused by heap allocation beyond the data break or growth of the current *stak* item beyond the data break), grows the data segment with *sbrk* (2) by *brkincr* bytes, and returns control, thus resuming the faulting instruction.

2.2. No use of C library

The shell makes no use of the C library beyond system calls, perhaps because the C library was not well-developed when the shell was written, perhaps to make the shell self-contained, or perhaps to avoid dangerous interactions among the shell, the shell's *malloc*, the C library, and the C library's *malloc* (3). The effect has been to make the shell fragile and less portable than if it did rely on the C library. For example, the shell's memory allocator does not co-exist with the C library, so it is not safe to call the *directory* (3) directory-reading routines, which call *malloc* (3).

3. Implementation Problems

3.1. Memory Management

The heap allocator, in `blok.c` (a modified V7 *malloc* (3)), and the *stak* allocator, in `stak.c`, together form the shell's memory allocator. They are intimate with each other's internals, in part because the heap allocator must move the top item on the *stak* when allocating heap storage, in order to keep the top item of *stak* storage at the top of the data segment.

alloc moves the top *stak* item to above the new top of the heap arena when the arena is grown, and promotes other *stak* items to *free*able permanent storage, chained together. `blok.c` rounds the number of bytes to allocate down by anding it with the complement of (*brkincr* minus one); this only works correctly if *brkincr* is a power of 2, yet `stak.c` adds 256 to *brkincr*, which starts off at 512! Thus the rounded-down value will be too large, which only hurts performance, fortunately. `stak.c` should probably double *brkincr* instead.

The shell's allocator cannot co-exist with some *malloc* implementations because it assumes that only it allocates storage, and some *malloc*s do likewise. [Korn1985] Further, the shell contains *malloc* and *free* definitions, but not a *realloc* definition, so uses of *realloc* in the C library will drag in the C library's *realloc*, which will refer to the wrong *malloc* and *free*. More subtly, because the old shell allocates storage above its data break,‡ even a tolerant *malloc* (3) which tried to co-operate with programs which do their own allocation via *brk* (2) or *sbrk* (2) would be misled and would likely step on the old shell's storage above its data break. Perhaps due in part to this problem, the old shell goes to great pains to avoid using the C library, except to execute system calls, necessitating reinvention of parts of it, notably the string functions. The new shell relaxes this restriction and so can use the C library freely.

The assumption that faulting instructions can be (and are) restarted by return from the appropriate signal handler does not hold on all machines of interest. In particular, the Sun-2, Sun-3, [Shannon1988] and Cray-1 kernel-and-hardware combination are known not to correctly restart faulting instructions, which can lead to failure to initialise memory, and thus to use of the address zero. Furthermore, on the Sun-3, and other machines which do not permit reference to address zero, the old shell's naive assumption that growing the heap will cure a memory fault does not hold for references to address zero, and the strategy of growing the heap on each fault merely grows the heap indefinitely, often until swap or page space is exhausted, when the old shell's memory allocator blows an assertion and dumps core, slowly.

Unfortunately the `stak.h` macros, especially *pushstak*, were not used everywhere that they should have been used, and in places the equivalent code was written out in-line, or other internal programming conventions were violated. Cray Research found and fixed the most troublesome of the abuses of *pushstak*, and these fixes found their way into the Ninth Edition shell. We found and fixed the rest.

3.2. Here Documents

Here documents are a means of supplying standard input to a command from a script and are denoted by '<<'.

Here documents appear to have been added to the original Seventh Edition shell at the last moment. The code is localised, but careless about error-checking and performance. Here documents are implemented by copying lines from the shell's input to a

‡ The *data break* or just *break* is the address of the lowest-numbered byte of the data segment not allocated to a UNIX process. There may be accessible memory between the break and the bottom of the stack segment, but touching it is bad form and may result in a memory fault ("segmentation violation").

temporary file during parsing until a line containing only the delimiter is seen; then later, during execution of the command, if the delimiter was not quoted, copying the first temporary file to a second temporary file while processing macros (e.g. expanding `$DMD`). If the second temporary file was created, it is opened as the command's standard input and unlinked, so it will not have to be removed later; otherwise the first temporary file will be opened as the command's standard input. In either case, the first temporary file will have to be unlinked later, but the shell may not get the chance if it is killed first or if the block containing the command with the here document was terminated via the `exec` built-in command, which replaces the shell with the command.

write system calls to the first temporary file were unchecked, so creating a here document when the file system containing `/tmp` is full may lead to odd behaviour and no diagnostic from the old shell. (*writes* to the second temporary file use a more general mechanism, *flush* in `macro.c`, and are still unchecked in the new shell. Oops!) Also, in the old shell, when copying input to the first temporary file, lines are collected until at least CPYSIZ (512) bytes are present, then are written to disk as whole lines, so CPYSIZ+ε bytes are written at a time, causing *write*s to be unaligned with file system block boundaries, and contributing to the slowness of here documents and thus to the slowness of unbundling of shell "bundles" or "archives". [Kernighan1984]

3.3. Directory Reading and Wildcard Expansion

The Seventh Edition shell reads directories to expand wildcards ("generate filenames"), using the *read* system call to read 16 bytes at a time and assuming a Seventh Edition directory layout. The Ninth Edition (and presumably System V Release 2) shell used some of the *directory* (3) routines from the C library, but used private versions of others. This was done so that memory allocation would be under the control of the shell's private *opendir* and *closedir*. Unfortunately, it did require the shell to know details of buffer allocation in the *directory* (3) routines, and those details changed between 4BSD and SunOS 3.0, for example. (We believe 4BSD used a static buffer but SunOS 3.0 allocates the buffer dynamically.)

3.4. I/O Redirection

When the old shell executes a redirected built-in command such as `set`, it saves the redirected descriptor by using *dup2* (2) to make a duplicate descriptor, on a fixed descriptor, USERIO, which is typically 10 and must be above the shell's user-accessible descriptor range (0-9). Unfortunately, the old shell isn't prepared to deal with multiple redirectors of built-in commands, so `set </etc/passwd >/dev/null` causes `set` to execute and then causes the old shell to read `/etc/passwd`(!).

When applied to any command, built-in or not, `<''` and `>''` have no effect in the old shell. This appears to be a relic from the days before `$*` and `$@`.†

† This undocumented misfeature of the shell was discovered by a naive and serendipitous user who was pleasantly surprised to find that `<$1` in a shell script 'did the right thing' whether the script was invoked with no arguments or with one, and commented upon this surprise.

3.5. Name-to-i-number translations

The code to run down `$PATH` looking for a command executed more system calls which translate file names to i-numbers than necessary; upon finding a command, it would *access* (2) the file, then *stat* (2) it.

3.6. Exit

On many (by intent, all) UNIX systems, a program which does not use the standard I/O library (*stdio*) [Kernighan1979] will not cause any part of *stdio* to be loaded with it. This is not true on SunOS 3.0, for example; a program such as the shell which does not use *stdio* still gets some of *stdio* loaded with it, due to *exit* calling *fflush*, and that in turn causes some *malloc* to be loaded. Sun's *malloc* includes 8,192 bytes of BSS (uninitialised data segment) containing its initial free block headers. This seems excessive, given that programs often use *malloc* in an attempt to *conserve* memory.

4. Fixes in the new shell

4.1. Memory Management

Our original fix to the memory allocator, to make it co-exist with the C library and work in general, was to delete `blok.c` thus invoking *malloc* (3) and to rewrite `stak.c` from scratch to use *malloc* (3). Much later we discovered that an alternative, less clean and less robust fix is to just delete private *directory* (3) functions, make *chkid* reject zero addresses, and provide a private *realloc* in `blok.c` which implements the semantics of *realloc* (3) using the private *malloc* and *free*, though one must also increase the values of `BRKINCR` and `BRKMAX` to at least the page size on some systems. This apparently works by causing every memory fault to increase the data segment enough to cover the largest allocation request normally seen inside the shell. The new shell does not use this fix.

The new shell uses *pushstak* and the other interface macros and functions where needed, and *pushstak* now arranges to grow the top item of *stak* storage as needed. The performance impact of this has not been measured, but appears to be insignificant; in any case, this checking is necessary. `stak.c` has been completely rewritten from scratch (see the Appendix).

We now simulate the single pointer to several interwoven stacks of *stak* storage by attaching a pointer to the previous *stak* item to each new *stak* item as it is allocated, and retaining one watermark pointer per stack.

We layer another function (*shfree*), on top of *free* (3), and the new shell is compiled with `#define free shfree` and without the old shell's `#define alloc malloc`, *alloc* being the name by which the heap allocator is invoked. *shfree* rejects attempts to free the address zero, addresses in the stack segment or of *stak* storage; *free* (3) is used directly to free *stak* storage. To distinguish heap storage from *stak* storage, the new shell's allocator attaches an integer containing a magic number, different for heap and *stak* storage, to each item of storage allocated. This costs a little bit of memory, but experiments on a PDP-11 suggest that this is not a serious problem.

We simply use *malloc* (3) and related functions, and thus require none of the complicated machinery for restarting faulting instructions. This has the pleasant side-effect

that a buggy shell wielding a wild pointer typically dumps core immediately, instead of growing its stack segment until the kernel kills it (producing a multi-megabyte core dump) minutes later.

4.2. Here Documents

We have repaired both of the here document bugs, with one minor loss of generality: the here document delimiter must not exceed CPYSIZ bytes. CPYSIZ bytes are now written to the first temporary file (until end-of-file is read), and the remaining fractional line is copied back to the start of the copying buffer, which is 2*CPYSIZ bytes long.

4.3. Directory Reading and Wildcard Expansion

We solved the messy problems of reading directories by deleting the private functions and using only the C library *directory* (3) functions to read directories. The shell was modified to call the new function *openqdir* instead of calling *opendir* directly; *openqdir* passes to *opendir* a copy of the file name with the 0200 bit, used by the shell internally to mark quoted characters (e.g. the first character of a command argument such as \?*), stripped from each character.

We also discovered that the code that implemented negated character classes (for example, `[!a-z]`) in the old shell was incorrect and had only worked by chance; Henry Spencer replaced the incorrect code with robust, working code.

4.4. I/O Redirection

The new shell is prepared to save multiple standard descriptors by duplicating them to whichever descriptors above the normal range are free.

I/O redirections now behave as one would expect: since the empty filename refers to the current directory, `<''` will open the current directory on some systems, and `>''` will, one hopes, fail with an error message.

4.5. Name-to-i-number translations

Use of *access* is inappropriate in the shell, as one wants to check against the shell's effective ids, and unnecessary, as one can easily check the permission bits obtained from the *stat*. This is faster because each system call, such as *access* or *stat*, which takes a filename as a parameter must translate it to the (device-number, i-number) pair used internally by UNIX to refer to files. This translation is relatively slow because it typically requires disk accesses, even on systems with *namei* caches. Still faster would be to simply try to *exec* (2) each filename in turn, and examine *errno* afterward; this will not work for the *type* (a.k.a. *whatis*) built-in, though, and we have not done this.

4.6. Exit

malloc is not an issue in the new shell, but unwanted static *stdio* buffers do take quite a bit of data space. Defining `exit(n) { _exit(n); }` to avoid *stdio* significantly reduces the shell's size, thus reducing the time needed to *fork(2)* and speeding command execution.

5. Methods

Debugging the shell is more difficult than one might expect. Initial debugging was largely by inspired guesses and tedious experimentation, due to the difficulty of examining multi-megabyte core dumps, which tend to be uninformative anyway.

Once the shell was made to stop catching SIGSEGV, it was possible to use debuggers to examine core dumps produced by buggy shells and produce stack traces, but lacking a truly useful debugger (such as *pi* (9.1)), [Cargill1986] we resorted, in the main, to printing interesting variables (with the shell's *prs* and *prn*, not *printf* (3)) and thinking about the output. One other helpful technique was to insert magic numbers into each instance of each relevant data structure when the instance was created, then check periodically for the presence of the number, and clear the number upon destruction of the instance. This simple technique alone was a great help in keeping the shell sane by detecting corruption and confusion early. We also linked the shell with a debugging version of *malloc* supplied by Sun (`/usr/lib/debug/malloc.o`), which checks the arena for consistency.

6. Conclusions

Debugging would certainly have been easier if the assumptions in the old shell code had been documented; we hope that this paper will save shell maintainers many hours. The new shell appears to be quite portable and has been run on the DEC PDP-11 under V7, Sun-3 under SunOS 3.x, Sun-4 under SunOS 4.0, and MIPS M/1000.

Our new shell now contains comments which describe most of the newly-discovered assumptions which had been hidden in the old shell.

Unfortunately, this version is not generally available, as it is derived from Ninth Edition code. The new version of `stak.c`, however, is not licensed and is reprinted in the Appendix to this paper.

7. Acknowledgements

Henry Spencer of the University of Toronto's Department of Zoology hired the author to perform the work described above, commented on drafts of this paper, ran various versions of the new shell as `/bin/sh` on his machines while we discovered new undocumented properties of the shell, and was patient while bugs were found and fixed. The Department funded this work.

Cray Research found some of the places in which *pushstak* should have been used in the old shell and repaired them, and fixed *pushstak*, in order to make the shell run on Crays, at least some models of which are incapable of restarting instructions which abort due to memory faults.

Dennis Ritchie incorporated Cray's fixes into the Ninth Edition shell, and urged the author to continue attempting to fix the shell rather than throwing it out and reimplementing it. (He was of course right, in part due to the subtleties of getting details such as quoting and *eval* just right. Nevertheless, a future reimplementation of the shell would benefit by using more of the tools available in the C library and elsewhere, possibly including *yacc* and *lex*.) Dennis also provided very helpful comments on a draft of this paper.

Ian Darwin, Beverly Erlebacher and Tom Glinos proof-read drafts of this paper and contributed helpful suggestions.

Any errors remaining in this paper are the responsibility of the author.

References

Bourne1978.
S. R. Bourne, ''UNIX Time-Sharing System: The UNIX Shell,'' *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1971-1990, 1978.

Cargill1986.
T. A. Cargill, ''The Feel of Pi,'' *Proc. Winter Usenix Conf. 1986*, 1986.

Johnson1978.
S. C. Johnson and D. M. Ritchie, ''UNIX Time-Sharing System: Portability of C Programs and the UNIX System,'' *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 2021-2048, 1978.

Kernighan1979.
Brian W. Kernighan and Dennis M. Ritchie, ''UNIX Programming – Second Edition,'' *UNIX Programmer's Manual, Seventh Edition*, January, 1979.

Kernighan1984.
Brian W. Kernighan and Rob Pike, *The UNIX Programming Environment,* Prentice-Hall, 1984.

Korn1985.
David G. Korn and Kiem-Phong Vo, ''In Search of a Better Malloc,'' *Proc. Summer Usenix Conf. 1985*, pp. 489-506, June 1985.

Ritchie1987.
Dennis Ritchie, *private communication*, 1987.

Shannon1988.
Bill Shannon, *private communication*, November, 1988.

Appendix: the new stak.c, minus debugging #ifdefs

This code was all written by the author; it is not subject to any source licences.

```
/* replaces @(#)stak.c1.4 */

/*
 * UNIX shell
 *
 * Stacked-storage allocation.
 *
 * Maintains a linked stack (of mostly character strings), the top (most
 * recently allocated item) of which is a growing string, which pushstak()
 * inserts into and grows as needed.
 *
 * Each item on the stack consists of a pointer to the previous item
 * (the "stakbsy" pointer; stakbsy points to the top item on the stack), an
 * optional magic number, and the data. There may be malloc overhead storage
 * on top of this.
 *
 * Pointers returned by these routines point to the first byte of the data
 * in a stack item; users of this module should be unaware of the "stakbsy"
 * pointer and the magic number. To confuse matters, stakbsy points to the
 * "stakbsy" linked list pointer of the top item on the stack, and the
 * "stakbsy" linked list pointers each point to the corresponding pointer
 * in the next item on the stack. This all comes to a head in tdystak().
 *
 * Geoff Collyer
 */

/* see also stak.h */

#include "defs.h"

#undef free                     /* refer to free(3) here */

#define STMAGICNUM 0x1235       /* stak item magic */
#define HPMAGICNUM 0x4276       /* heap item magic */
#define MAGICSIZE BYTESPERWORD          /* was once zero */

/* imports from libc */
extern char *malloc(), *realloc();
extern char *memcpy(), *strcpy();

/* forwards */
char *stalloc(), *growstak(), *getstak();

unsigned brkincr = BRKINCR;             /* used in stak.h only */

static char *
tossgrowing()                   /* free the growing stack */
{
        if (stakbsy != 0) {             /* any growing stack? */
                register struct blk *nextitem;

                /* verify magic before freeing */
                if (((int *)Rcheat(stakbsy))[1] != STMAGICNUM)
                        error("tossgrowing: bad magic on stack");
                ((int *)Rcheat(stakbsy))[1] = 0;        /* erase magic */

                /* about to free the ptr to next, so copy it first */
                nextitem = stakbsy->word;
                free((char *)Rcheat(stakbsy));
                stakbsy = nextitem;
        }
}

static char *
stalloc(asize)                  /* allocate requested stack space (no frills) */
int asize;
{
        register char *newstack;
        register int size = asize;

        newstack = malloc((unsigned)(sizeof(struct blk) + MAGICSIZE + size));
        if (newstack == 0)
                error(nostack);

        /* stack this item */
        *((struct blk **)Rcheat(newstack)) = stakbsy; /* point back at old stack top */
        stakbsy = (struct blk *)Rcheat(newstack);     /* make this new stack top */
        newstack += sizeof(struct blk);         /* point at the data */

        /* add magic number for verification */
        *((int *)Rcheat(newstack)) = STMAGICNUM;
        newstack += MAGICSIZE;
        return newstack;
}

static char *
grostalloc()                    /* allocate growing stack */
{
        register int size = BRKINCR;

        /* fiddle global variables to point into this (growing) stack */
        staktop = stakbot = stakbas = stalloc(size);
        stakend = stakbas + size - 1;
}

/*
 * allocate requested stack.
 * staknam() assumes that getstak just realloc's the growing stack,
 * so we must do just that. Grump.
 */
char *
getstak(asize)
int asize;
{
        register char *newstack;
        register int staklen;

        /* + 1 is because stakend points at the last byte of the growing stack */
        staklen = stakend + 1 - stakbas;        /* # of usable bytes */
        newstack = growstak(asize - staklen);           /* grow growing stack to asize */
        grostalloc();   /* allocate new growing stack */
        return newstack;
}

/*
 * set up stack for local use (i.e. make it big).
 * should be followed by 'endstak'
 */
char *
locstak()
{
        if (stakend + 1 - stakbot < BRKINCR)
                (void) growstak(BRKINCR - (stakend + 1 - stakbot));
        return stakbot;
}

/*
 * return an address to be used by tdystak later,
 * so it must be returned by getstak because it may not be
 * a part of the growing stack, which is subject to moving.
 */
char *
savstak()
{
        assert(staktop == stakbot);     /* assert empty stack */
        return getstak(1);
}

/*
 * tidy up after 'locstak'.
 * make the current growing stack a semi-permanent item and
 * generate a new tiny growing stack.
 */
char *
endstak(argp)
register char *argp;
{
        register char *oldstak;

        *argp++ = 0;  /* terminate the string */
        oldstak = growstak(-(stakend + 1 - argp));    /* reduce growing stack size */
        grostalloc();   /* alloc. new growing stack */
        return oldstak; /* perm. addr. of old item */
}

/*
 * Try to bring the "stack" back to sav,
 * and bring iotemp's stack back to iosav.
 */
tdystak(sav, iosav)
register char *sav;             /* returned by growstak(): points at data */
register struct iono d *iosav;  /* an old copy of iotemp (may be zero) */
{
        rmtemp(iosav);                  /* pop temp files */
        if (sav != 0 && ((int *)Rcheat(sav))[-1] != STMAGICNUM) /* sav -> data */
                error("tdystak: bad magic in argument");

        /*
         * pop stack to sav (if zero, pop everything).
         * sav is a pointer to data, not magic nor stakbsy link.
         * stakbsy points at the ptr before the data & magic.
         */
        while (stakbsy != 0 && (sav == 0 ||
          (char *)stakbsy != sav - sizeof(struct blk) - MAGICSIZE))
                tossgrowing();          /* toss the stack top */
        grostalloc();           /* new growing stack */
```

```
}

stakchk()                       /* reduce growing-stack size if feasible */
{
        if (stakend - staktop > 2*BRKINCR) /* lots of unused stack headroom */
                (void) growstak(-(stakend - staktop - BRKINCR));
}

char *                  /* address of copy of newstak */
cpystak(newstak)
char *newstak;
{
        return strcpy(getstak(strlen(newstak) + 1), newstak);
}

char *                  /* new address of grown stak */
growstak(incr)          /* grow the growing stack by incr */
int incr;
{
        register char *oldbsy;
        unsigned topoff, botoff, basoff;
        int staklen;

        if (stakbsy == 0)       /* paranoia */
                grostalloc();   /* make a trivial stack */

        /* paranoia: during realloc, point at previous item in case of signals */
        oldbsy = (char *)stakbsy;
        stakbsy = stakbsy->word;

        topoff = staktop - oldbsy;
        botoff = stakbot - oldbsy;
        basoff = stakbas - oldbsy;

        /* + 1 is because stakend points at the last byte of the growing stack */
        staklen = stakend + 1 + incr - oldbsy;

        if (staklen <= sizeof(struct blk) + MAGICSIZE)      /* paranoia */
                staklen = sizeof(struct blk) + MAGICSIZE;

        if (incr < 0) {
                /*
                 * V7 realloc wastes the memory given back when
                 * asked to shrink a block, so we malloc new space
                 * and copy into it in the hope of later reusing the old
                 * space, then free the old space.
                 */
                register char *new = malloc((unsigned)staklen);

                if (new == NIL)
                        error(nostack);
                (void) memcpy(new, oldbsy, staklen);
                free(oldbsy);
                oldbsy = new;
        } else {
                /* get realloc to grow the stack to match the stack top */
                if ((oldbsy = realloc(oldbsy, (unsigned)staklen)) == NIL)
                        error(nostack);
        }
        stakend = oldbsy + staklen - 1;         /* see? points at the last byte */
        staktop = oldbsy + topoff;
        stakbot = oldbsy + botoff;
        stakbas = oldbsy + basoff;

        /* restore stakbsy after realloc */
        stakbsy = (struct blk *)Rcheat(oldbsy);
        return stakbas;         /* addr of 1st usable byte */
}

/* ARGSUSED reqd */
addblok(reqd)                   /* called from main at start only */
unsigned reqd;
{
        if (stakbot == 0)       /* called from main, 1st time */
                grostalloc();   /* allocate initial arena */
        /* else won't happen */
}

/*
 * Heap allocation.
 */
char *
alloc(size)
unsigned size;
{
        register char *p = malloc(MAGICSIZE + size);

        if (p == NIL)
                error(nospace);

        *(int *)Rcheat(p) = HPMAGICNUM;
        p += BYTESPERWORD;              /* fiddle ptr for the user */
        return p;
}

/*
 * the shell's private "free" - frees only heap storage.
 * only works on non-null pointers to heap storage
 * (below the data break and stamped with HPMAGICNUM).
 * so it is "okay" for the shell to attempt to free data on its
 * (real) stack, including its command line arguments and environment,
 * or its fake stak.
 * this permits a quick'n'dirty style of programming to "work".
 * the use of sbrk is relatively slow, but effective.
 */
shfree(p)
register char *p;
{
        extern char *sbrk();

        if (p != 0 && p < sbrk(0)) {            /* plausible data seg ptr? */
                register int *magicp = (int *)Rcheat(p) - 1;

                /* ignore attempts to free non-heap storage */
                if (*magicp == HPMAGICNUM) {
                        *magicp = 0;  /* erase magic */
                        p -= BYTESPERWORD; /* get orig. ptr back */
                        free(p);
                }
        }
}
```

Job and Process Recovery in a UNIX–based Operating System

Brent A. Kingsbury
John T. Kline

Cray Research, Inc.
1440 Northland Drive
Mendota Heights MN 55120
brent@yafs.cray.com
jtk@hall.cray.com

ABSTRACT

Many computationally-intense programs require more execution time than the underlying system can deliver in a single continuous time interval; yet preventive maintenance, reconfiguration, and new kernel development requirements sometimes force the need for scheduled downtime. Also, system crashes due to power failures and other causes result in additional downtime. On generally available UNIX[1] systems (such as Berkeley and AT&T System V), this downtime results in the loss of work in progress at the time of system shutdown.

To greatly reduce these losses and allow the execution of very long running programs, we have extended the kernel to directly support the *checkpointing* and *restarting* of processes and related collections of processes (jobs). The kernel extensions in combination with a batch job daemon provide automatic and transparent recovery of batch jobs after scheduled shutdowns and, in some cases, system crashes. In addition, commands exist enabling users to interactively checkpoint and restart processes.

1. Introduction

In 1982, Cray Research began work on a new operating system. This operating system is today known as UNICOS,[2] and is derived from the AT&T UNIX System V operating system; it is also based in part on the Fourth Berkeley Software Distribution. As previous operating systems available on Cray machines such as CTSS [1] and COS[3] have long supported a recovery facility, and since AT&T's UNIX System V has no such facility, we felt it necessary to add a recovery capability to the UNICOS operating system.

Our efforts have resulted in the creation of a transparent recovery facility based on two new system calls named **chkpnt** and **restart**. These system calls along with enhancements to NQS,[4] the UNICOS batch subsystem, provide an automatic and transparent batch job recovery facility for scheduled shutdowns and, in some cases, system crashes. Additionally, the user commands **chkpnt** and

[1] UNIX is a registered trademark of AT&T.

[2] UNICOS is a registered trademark of Cray Research, Incorporated.

[3] COS refers to the *Cray Operating System*, as developed by Cray Research, Incorporated.

[4] NQS refers to the *Network Queueing System* as originally developed by Sterling Software for the *NASA Ames Research Center*.

restart allow users to explicitly checkpoint and restart their own processes.

Since the UNICOS recovery facility has been implemented to be completely transparent to the affected processes and jobs, no modifications are necessary for the majority of applications to be checkpointed and later restarted. Unless a target process takes specific steps to determine that it has been checkpointed and restarted, it will be unaware that such activity has taken place.

Readers who are familiar with the concepts of process migration in distributed operating systems such as LOCUS [2] or the V-system [3], and in networked collections of independent UNIX systems [4,5,6,7] will note some similarities with the work discussed in this paper. Considering that the movement of a process between machines requires that the process be resurrected at the destination machine, some implementations of process migration are, to varying degrees, implementations of a checkpoint and restart facility.

Unlike those systems, however, our goal was to provide an extremely transparent recovery facility in which entire collections of related processes could be checkpointed and restarted as a single consistent unit. Furthermore, the UNICOS recovery facility was designed to support the recovery of processes using pipes, unlinked files, and partially completed asynchronous I/O operations.

During this effort, we encountered several interesting problems, and one fundamental limitation. The remainder of this paper describes our work.

2. Definitions

In UNICOS, a *job* refers to a collection of related processes that the system tracks as a single unit for the purposes of resource control, accounting, and recovery. A job is created when a process running as *root* performs a **setjob** system call (**setjob** is a UNICOS extension not found in UNIX), assigning a unique *job–ID* to the calling process, and all of its subsequent descendant processes. This typically occurs when a user logs into the system, and when an NQS batch request is spawned. Unlike *process group–IDs*, however, a process cannot change its *job–ID* unless it runs as root.

A *multitask group* is defined as a collection of tightly coupled processes providing multiple threads of execution that share several resources, including open file descriptors and a common address space.

A *restart file* is a file containing all of the information needed to restore the process, multitask group, or job to its execution state at the time when the restart file was created.

The **chkpnt** system call is responsible for checkpointing a process, multitask group, or job. This is accomplished by first "freezing" the target processes, and then creating the restart file containing the information needed to restore the processes to execution at a later time.

The **restart** system call can be thought of as the inverse of the **chkpnt** system call. The **restart** system call accepts a restart file as input, and then restores the process, multitask group, or job defined therein to its original execution state.

Many of the constraints and conditions governing the successful checkpoint and recovery of a process, multitask group, or job are enforced regardless of the process grouping being referenced. Thus, the phrase *recovery of a process* is used to describe the recovery of a process, multitask group, or job, unless otherwise indicated.

Similarly, the phrase *checkpoint of a process* is used to indicate the checkpoint of a process, multitask group, or job, unless otherwise noted.

3. Requirements

In order to produce an effective recovery facility, we had to meet several requirements. This section describes the requirements and explains why they were adopted.

3.1. Transparent Recovery

In some instances, a user may have access to the binary image of an application but may not have access to the original source code. If the recovery facility requires modification of such applications, these applications cannot be checkpointed and restarted.

Even when the original source code of a program is available, the complexity of the program may make modifications undesirable for the sake of recovery. With new programs being written every day, any required modification for recovery would make the already burdensome work of porting software even more difficult. For these reasons, the UNICOS recovery facility had to be implemented so that all recovery activity would be transparent to the target processes.

We realized, however, that some programs would need to recognize when recovery activity had taken place. For example, if a screen–oriented text editor were restarted, the editor would need to determine the user's terminal characteristics and repaint the screen accordingly. In UNICOS, this is accomplished by sending all restarted processes a special dedicated signal (ignored by default, in keeping with the conventions of a transparent recovery facility) indicating that the processes have been restarted.

3.2. User-directed Recovery

At any time, users must be able to checkpoint and restart their own processes. There are several benefits to supporting these capabilities.

First, if a user observes that a program is producing errant results but he or she does not have time to debug the program at that moment, he or she may checkpoint the process running the errant program. At a more convenient time, the user can restart the process from where it left off, under the control of a debugger, and determine the cause of the failure.

Second, although the UNICOS recovery facility provides automatic recovery of batch jobs across controlled system shutdowns, it does not periodically checkpoint jobs in progress as a protection against system crashes. Allowing users to explicitly checkpoint and restart their own processes enables them to protect their work by periodically checkpointing their own programs. If a system crash occurs, users can restart from the most recent restart file defining their program.

Third, any checkpoint operation consumes both system time and disk space. If a process to be checkpointed is 60 megawords (480 megabytes) in size, the restart file created for that process will be similarly large. Since users are in the best position to know when their program's execution warrants economical checkpointing, we wish to give them as much control as possible.

Finally, except for one case, we do not allow users to checkpoint and restart entire jobs. In UNICOS, the *job* is the first line of resource control defense. If we allow users to checkpoint and restart entire jobs without restriction, users can submit NQS batch requests, checkpoint the jobs created to satisfy the requests, and interactively restart the jobs. Allowing such activity would give users the ability to obtain more than their fair share of the machine.

As the only exception to the prohibition against users checkpointing and restarting entire jobs, a command called **qchkpnt** exists. When executed as part of a batch job shell script, **qchkpnt** causes NQS to checkpoint the running job. If the system subsequently crashes, NQS will attempt to restart the job from the most recently created restart file.

3.3. System Administrator and NQS Directed Job Recovery

To provide recovery of batch jobs across scheduled shutdowns, a privileged NQS daemon process running as root must be able to checkpoint and restart an entire job. There are also some situations in which it is useful for a system administrator running as root to checkpoint or restart an entire job.

The ability to checkpoint and restart an entire job is provided because many jobs consist of a group of cooperating processes that cannot be individually checkpointed and restarted in a consistent manner. For example, a sender process may be sending its results to a filter process via a pipe. We cannot checkpoint and restart the sender or the filter process by themselves, since it would be impossible to preserve the cooperative relationship between the two.

3.4. System Security

It must not be possible to fool the system into restarting a process from a corrupt or fake restart file, nor must it be possible to read information from a restart file that would compromise the integrity of the system.

Since a restart file describes information that, if altered, could grant unauthorized access to the machine (for example, *user–ID* information), restart files must be carefully protected against modification and other trickery. Also, if any *setuid* processes are contained in a restart file, the restart file must not be readable, because sensitive information could be revealed.

3.5. Kernel Reconfiguration Independence

It must be possible to reconfigure the system with differing quantities of kernel resources, without losing the ability to restart previously checkpointed processes. This means that if we checkpoint a process in the middle of a system call, the proper restart of that system call must not depend upon kernel data structure addresses remaining unchanged. However, this requirement does not extend to cover new releases of the operating system.

3.6. Slow I/O Recovery

When checkpointing a process, it is feasible to wait for fast I/O operations (for instance, disk I/O), to complete before recording the state of the process for its eventual recovery. However, slow I/O operations, such as those on a pipe or tty may not complete any time soon. Since we must checkpoint the process in a timely manner, we cannot wait for slow I/O operations to finish.

In our original implementation, we "solved" this problem by not recovering slow I/O requests. Therefore, a read operation on a pipe or tty returned the error code of *EINTR* when the process was restarted, indicating that the system call was interrupted and should be re-executed by the caller. In so doing, we placed the burden of restarting slow I/O operations on the application instead of the kernel.

Unfortunately, a surprising number of programs were found that did not properly reissue I/O requests for slow devices when the requested I/O operation failed with the *EINTR* error return. Worse yet, some programs that ordinarily handled the *EINTR* case quite well, failed when they were checkpointed and restarted.

One notable example of this problem occurred with the Bourne shell, which, when checkpointed under our original implementation, always exited upon restart if it had been reading input from the user's terminal. What we discovered was that the simple *EINTR* return from the **read** system call was not sufficient. The Bourne shell demanded a signal to properly recognize the *EINTR* error return.

For these reasons, it became a requirement to recover all slow I/O operations, so that even when the application failed to understand the *EINTR* error return, it would still be restarted correctly.

3.7. Swap Activity Independence

Unlike the recovery facilities of the CTSS and COS operating systems, the UNICOS recovery facility must not depend on swap activity. In UNICOS, processes rather than entire jobs are swapped when main memory becomes scarce. Thus, no complete job image can ever be guaranteed to be totally contained either in swap space or in memory. Also, the increase in the main memory size of our computers has reduced the level of swap activity, making dependence on such activity undesirable.

Finally, kernel state information that is never swapped exists for each process in the system. Numerous examples of such information abound, including I/O request structures, shared file table structures, pipe data, file locks, and process table structures.

3.8. Accounting Information Accuracy

Since we designed the recovery of processes to be as transparent as possible, many accounting statistics must be preserved. In particular, system calls that return CPU timing information must return results as though no recovery activity had taken place; otherwise, timing statistics gathered by a particular program will be inaccurate. This requirement resulted in changes to the implementation of the **times** system call, and other timing system calls in UNICOS.

Resource quota and usage information must also be preserved, which becomes important in the context of NQS. When a batch job is spawned by NQS, it is given a set of resource quota limits. If the job is checkpointed and restarted during the course of its execution, its original limits and the totals of the resources already consumed must be preserved if there is to be reliable control over machine

resource usage.

As a final concern, accounting statistics used for the purposes of billing must not be adversely affected by recovery activity; at many sites, users are billed for their consumption of machine resources. As we preserve information such as CPU usage statistics, we must be careful to not write process accounting records that bill users incorrectly for their use of the machine.

3.9. Identifier Preservation

During execution, a collection of processes may acquire knowledge about their peer processes and communicate with each other using *process–ID*, *process group–ID*, or *job–ID* values. If processes are restarted without preserving these *ID* values, it becomes impossible for processes to reliably communicate and otherwise keep track of one another. The changing of *ID* values also creates new problems when accounting for the various machine resources consumed by restarted processes.

For these reasons, the identifiers of *process–ID*, *process group–ID*, and *job–ID* are preserved across checkpoint and restart events. This policy creates a new problem, however, in that a needed *ID* value may already be in use by another process when a restart operation is attempted. This new problem remains unsolved, and is discussed later in this paper.

The preservation of *ID* values also creates the constraint that the **restart** system call can never be implemented to be directly analogous to an **exec** system call. If **restart** were analogous to **exec**, we could no longer preserve the various process identifiers of the recovered processes. Therefore, a **restart** system call must be implemented somewhat like a **fork** system call, except that an unknown number of descendants are created, which in turn become the originally checkpointed processes.

4. Implementation

This section discusses the structure of a restart file, and the general algorithms of the **chkpnt** and **restart** system calls implementing UNICOS recovery.

4.1. Restart File Structure

A restart file is constructed whenever a process, multitask group, or job is successfully checkpointed, and contains the information needed to restore the checkpointed processes to their original state. As shown in Figure 1, a restart file is divided into several sections.

Section	Description
Header	Describes the remainder of the restart file
Inode descriptors	Describes each file in use
File table descriptors	Describes open file table entries
File lock descriptors	Describes all file locks in use
I/O request descriptors	Describes all I/O requests in progress
Object data	Pipe (FIFO) and unlinked file data
Process images	Actual images of process text and data regions

Figure 1: Restart File Structure

The *header* section of a restart file contains information describing the contents of the restart file. Fields in the header define the type of entity contained in the restart file (for example, a process or job), and the number of various object descriptions appearing in the file (for example, the number of inode, file table, file lock, and I/O request descriptors). Other information present in the header includes the number of processes and multitask groups defined in the restart file, and the characteristics of any original controlling tty connection.

The *inode descriptor* section of a restart file contains descriptors for each file in use by the checkpointed processes. Due to the impracticality of maintaining path information for each open file, we do not keep track of the open files by name. Instead, we record the mount device and inode number of

each file. This information later allows us to directly retrieve the inode describing the file, without encountering all of the problems of changed directory protections, and *setuid* considerations.

Additional information recorded for each open file, including the *inode generation number*, prevents files from being erroneously recovered. (When all links to an inode are removed, the generation number of the inode is incremented making it possible to determine that a file originally in use by a checkpointed process has been deleted.) Also, if the file refers to a pipe or an unlinked regular file, the offset within the restart file at which any associated data can be found is also present.

Each open file table entry in use by the checkpointed processes has a corresponding descriptor in the *file table descriptor* section of a restart file. A file table descriptor describes a possibly shared file table entry in use by one or more of the checkpointed processes. Each file table descriptor also identifies the inode descriptor of the file that is accessed.

File locks in use by the checkpointed processes have a corresponding descriptor in the *file lock descriptor* section of a restart file. Each descriptor identifies the inode descriptor of the file to which the lock was applied and contains a description of the original lock.

All I/O request structures in use by the checkpointed processes have a corresponding descriptor in the *I/O request descriptor* section of a restart file. Along with a compact description of the saved I/O request, each I/O request descriptor identifies the inode descriptor of the file for which the I/O is taking place, and any other dependent I/O request structures.

Any data present in any pipes or unlinked regular files referenced by the checkpointed processes is placed in the *object data* section.

The remainder of the restart file consists of the actual process images, along with their associated *process* and *user* structures.

4.2. The chkpnt System Call

The **chkpnt** system call accepts four arguments identifying the process or job to be checkpointed, the pathname of the restart file to be created, and an indication of whether or not the processes should be killed at the time of checkpoint. This latter capability is important, because a newly created restart file might be invalidated by the continued actions of the processes that it describes. For example, if a process is checkpointed but not killed, it will continue to execute when the checkpoint operation is complete. If the process goes on to modify its open files, it may make alterations inconsistent with its saved state in the restart file, and therefore invalidate the restart file.

The execution of a **chkpnt** system call procedes in two distinct phases, which are discussed below in the order that they occur.

4.2.1. Phase 1: Freezing the Target Processes

When the **chkpnt** system call is invoked, it scans the appropriate kernel tables to quickly identify the target processes to checkpoint. Ownership checks are then performed, ensuring that the caller owns the processes to be checkpointed. In keeping with UNIX convention, a super–user can checkpoint any process or job.

When ownership rights are verified, we begin *freezing* the target processes for checkpointing. We cannot simply copy the text and data of the target processes into the restart file, because the processes could change their state while we were making the copy.

Depending upon the circumstances, a process may be frozen quickly or slowly. The list of the possible cases is as follows:

1. *The target process is not executing a system call.*

   In this case, the target process can be frozen quickly. If the process is presently connected to a CPU executing user code, the CPU is immediately taken away, and the process is then blocked from further execution.

2. *The target process is executing a system call but is sleeping at an interruptable priority waiting for a kernel event to occur.*

Session Management in System V Release 4

Tim Williams

AT&T Bell Laboratories
Summit, New Jersey 07901

ABSTRACT

This paper describes the new session architecture and controlling terminal architecture in the System V Release 4.0 kernel designed to provide job control support, enhanced login session security and controlling terminal reusability, as well as maintain backwards compatibility with previous releases of System V. A subset of this work dealing with job control, has been incorporated into the POSIX 1003.1 standard and stems from the cooperative and exhaustive efforts of a POSIX working group made up of representatives from Hewlett-Packard, Berkeley, MIT, Usenix and AT&T. SVR4's session management model builds on that work.

1. Introduction

1.1 Background

Traditionally, System V has managed login sessions by providing the organizing principle of a *process group*. Process groups allow all of the processes belonging to a login session to be clearly identified and distinguished from all other processes in the system; events generated from controlling terminals such as "loss of carrier" (`SIGHUP`) or "user interrupt" (`SIGINT`) can then be directed to those processes.

The process that creates a process group, becomes the leader of the group, and is granted some distinct privileges in the group. It gives an identity to the group, since its process ID is allocated as a process group ID for subsequent members of the group. It serves as the ancestor of all members in the group, since the process group ID will be inherited by of its descendents that do not themselves create process groups. It may allocate a controlling terminal to the process group, thereby becoming that terminal's *controlling process*; when it does, the process assumes responsibility for management of the controlling terminal as a resource of the process group. The terminal will remain allocated to the process group until the process group leader exits, or until the device driver of the controlling terminal is closed.

This last privilege is important for programs such as `getty` and `ct` that are responsible for granting terminal access to users of the system. In addition, since the act of creating a process group has the side effect of divorcing a process from any existing affiliation with a login session, process groups also provide a convenient way of creating system daemons.

1.2 Limitations

Unfortunately, there are several limitations to System V's traditional process group model.

1. Should a controlling terminal be deallocated by closing the terminal's device driver, the process group leader may not allocate another; at best, the process must exit, the program be re-forked and re-made into another login session, making maintenance of state information between functionally similar login sessions difficult. It would be useful if a login session could serially allocate multiple controlling terminals over the course of the its lifetime. File descriptors created or inherited by session members to the original controlling terminal should be automatically redirected to a newly allocated controlling terminal transparently to the members of the login session.

2. If the controlling terminal senses "loss of carrier," it is no longer usable by the controlling process's process group; whether it remains allocated as a controlling terminal to the process group is typically a decision made by the controlling terminal's device driver, and has been a long standing source of inconsistency between implementations. It would be useful if a standard (and secure) way existed of reconnecting to terminals in this state.

3. The operating system provides no formal arbitration for processes in the process group that are simultaneously contending for use of their login session's controlling terminal. All members of the process group are given equal access to their controlling terminal and are left by the kernel to fend for themselves.

A limited form of order may be imposed by System V shells. Shells typically insulate the login session from processes started in the background by duping their standard input to `/dev/null`, preventing reads of the terminal via that file descriptor; similarly, they protect background processes from signals by setting them up to ignore the controlling terminal attention signals, `SIGINT` and `SIGQUIT`, which it assumes are meant for processes executing in the *foreground* of the controlling terminal. This method provides no arbitration for reads performed by background processes over new file descriptors, such as ones created by opening `/dev/tty`; nor does it provide arbitration for writes or ioctls from background processes.

Attempts to solve the problem with specialized controlling terminal device drivers, such as `/dev/sxt` have had limited utility, in general, because of their lack of support from and integration with the kernel. They fall far short of the success enjoyed by 4.3BSD's solution to the problem, job control. Yet although the functionality of 4.3BSD's job control is widely acclaimed, experience has pointed out problems with its implementation that prevented us from adopting it as is. Among them:

1. It lacks any real security afforded by System V's process group mechanism that would serve to isolate login sessions from one another. Processes that have read permissions for a controlling terminal may alter a terminal's process group to any other existing process group sharing the same user ID as that process. Worse yet, they may alter a terminal's process group to any non-existent process group; should that process group later be created, it will unwittingly inherit that controlling terminal.

2. It lacks the concept of a controlling process, the process that allocated the controlling terminal and will assume responsibility for it. A symptom of this flaw can be seen in the way the model treats the "loss of carrier" condition. `SIGHUP` is sent to the terminal's current process group, which may be a process ill-equipped to handle it (the process may be ignoring the signal, and not accessing its controlling terminal).

3. It lacks a consistent treatment of processes that access their controlling terminal from the background. A stop signal is sent to their process group; yet if they remain stopped when they are orphaned, they are continued (`SIGCONT`) and sent a hangup indication (`SIGHUP`) one at a time, on an individual basis. If they happen to be ignoring `SIGHUP`, they will be sent `SIGKILL` without warning the next time that they are about to stop, preventing them from performing any necessary cleanup.

1.3 Security Concerns

In addition, the present System V session model suffers from a few security problems.

1. Since a controlling process is given responsibility for management of controlling terminals, when that process exits, the controlling terminal should be deallocated and access to the controlling terminal by surviving processes in the session should end. In the past, System V attempted to accomplish this by sending `SIGHUP` to all members of the session. This was insufficient if processes are ignoring `SIGHUP` Should such processes continue to access the controlling terminal, they would interfere with later login sessions using the same terminal. At best, the next user of the terminal will receive apparently random output from the previous users; at worst, in the hands of a malicious user, the security of the next

user can be compromised.

2. If a member of a login session detaches itself from that login session by calling *setpgrp*(2), its controlling terminal will be relinquished, but any existing file descriptors to the terminal that had been inherited by the process when it was forked will remain active. Such a process is no longer under control of the terminal, and will not even be sent `SIGHUP` when the original login session is dissolved.

3. There is a well know window between the deallocation of a controlling terminal on exit of its controlling process and the reallocation by a respawned `getty` process, when another misbehaving process may *steal* away the terminal as its controlling terminal, preventing further login sessions on it.

4.3BSD effectively solves problems of controlling terminal security by invalidating indiscriminately every file descriptor in the system associated with a login session's controlling terminal when that login session ends. Unfortunately, this method introduces incompatibilities in existing System V applications, and inconsistencies with existing System V semantics. In addition to file descriptors that are part of the login session, it invalidates file descriptors that are outside the scope of the login session, even if they are owned by root or daemon processes or have been obtained legally via opens of the terminal driver that have passed existing security checks imposed by the file system. A new, but equally effective method is required for System V.

SVR4 will present solutions to both the limitation of the old model, and the security problems inherent within it, while remaining compatible with existing applications software.

2. A New Architecture

2.1 Design Principles

Four principles guided the design of the new session management architecture.

1. Maintain and enhance the role of the process that creates a login session as the leader of that session, in charge of and completely responsible for a session's security, integrity and resource management.

2. Maintain and enhance the role of file system permissions as the sole arbiter of physical access to terminals. For the most part, if a process successfully opens a terminal driver, the system should not later revoke permission to access it.

3. Incorporate the "look and feel" of 4.3BSD job control.

4. Eliminate existing inconsistent and unpredictable behavior.

2.2 Sessions

At the core of SVR4's new session management lies a POSIX compatible process group architecture, defined by the cooperative efforts of members of a POSIX job control working group chaired by Dave Lennert (then of Hewlett-Packard) and hosted as a set of teleconferences by AT&T.

POSIX job control solves the requirement of orderly arbitration of the controlling terminal among members in a session. It defines two attributes of a process, its *session* and its *process group*. Similarly, it defines two attributes of a controlling terminal, its *session* and its *foreground process group*.

Sessions have much of the quality of SVR3 process groups. They serve to organize a login session, insulating the member processes from all other login sessions active in the system. Their leaders have the same responsibilities as SVR3 process group leaders. A session leader's process ID is used as a *session ID*, a new process attribute used to identify the members of a session. They are responsible for allocating controlling terminals to the session, and for managing those controlling terminals once they are allocated to the session. In this model, the

session leader is in some sense, a trusted process. "loss of carrier" indications are delivered to the session leader, not the foreground process group, since they are considered the most reliable process to deal with the termination of the login session.

A session may have one or more process groups; a process group may have one or more processes. Processes may freely migrate from their current process group to any other process group within the same session, but they may not leave their session except by creating a new session, becoming that session's session leader and only member.

Process groups have much of the quality of 4.3BSD process groups, except that they must be created by a process group leader, the process with the same process ID as process group ID. This principle lends consistency to the allocation of IDs. Only the *fork*() system call may allocate a new ID, whether it be a process ID, process group ID or session ID. This policy contrasts with 4.3BSD, which effectively allocates process group IDs when is specifies a non-existent ID to the `TIOCSPGRP` ioctl; should a process with the same ID subsequently be forked, it unexpectedly inherits the controlling terminal.

Process groups are used to negotiate the simultaneous use of a controlling terminal by its session members. At any one time, only one process group, the current *foreground process group* is allowed unlimited access to the controlling terminal. All other process groups are considered *background process groups*, and are subject to a job control line discipline similar to 4.3BSD, which typically sends them a stop signal whenever they attempt access.

The architecture guarantees insulation of controlling terminals from processes outside of their session. A terminal's foreground process group can only be changed by a member of the terminal's session, and is always selected from among the existing process groups in the terminal's session. The architecture guarantees that all of the members of a process group strictly belong to only one session by placing restrictions on the creation of process groups and session. Since a process's ID is allocated as both a process group ID and a session ID when that process creates a new session, the system must disallow processes from becoming session leaders if their process ID is still in use as a process group ID by other members of its original session.

The architecture makes several assumptions about the orderly use of a controlling terminal.

1. In contrast to 4.3BSD, the smallest unit of control is the process group, not the process. Except for the "loss of carrier" indication, as detailed above, all events generated by the system to arbitrate controlling terminals are sent to all the members of a process group. When processes are continued by the system, their entire group is continued, not just specific processes in a group.

2. Again in contrast to 4.3BSD, stopped processes are not considered abandoned when they are orphaned. Since processes are always stopped with their groups, as long as there exist members in that process's process group that have parents in different process groups but in the same session (i.e. with the same controlling terminal), there exists the chance that those parents will take on the responsibility themselves of restarting the stopped process group. When this condition is no longer true (i.e. when the last such parent either exits, leaves the session or joins the process group of its child) the process group is considered to be an *orphaned process group*; the system will continue its process group and send its process group a hangup indication, implying that it has been detached from its terminal. Orphaned process groups are treated differently by the job control line discipline. If the system were allowed to stop them, there is no guarantee that they would ever be restarted; the controlling terminal therefore generates **EIO** errors when they attempt access from the background.

2.3 Controlling Terminals

2.3.1 Session Member Consistency The source of several of the limitations listed above in SVR3's model is rooted in its implementation of controlling terminals. Associated with each process is state information about that process's controlling terminal. It is initialized when the

```
Forever
        If this is not the calling process's controlling terminal
                Return ACCESS-ALLOWED

        If the calling process's session ID is not equal to the
        terminal's session ID, the session leader must have
        deallocated its controlling terminal
                If the process is not ignoring or holding SIGHUP
                        Send SIGHUP to its process group
                Else
                        Return ACCESS-ALLOWED

        Else, if the calling process is in the foreground
                Return ACCESS-ALLOWED

        Else, if the calling process is only reading terminal settings
                Return ACCESS-ALLOWED

        Else, if the calling process is reading from the terminal
                If the calling process is ignoring or holding SIGTTIN
                or if the calling process's group is orphaned
                        Return ACCESS-DENIED
                Else, send SIGTTIN to its process group

        Else, if the calling process is changing terminal settings
                If the calling process is ignoring or holding SIGTTOU
                        Return ACCESS-ALLOWED
                Else, if the calling process's group is orphaned
                        Return ACCESS-DENIED
                Else, send SIGTTOU to its process group

        Else, the calling process is writing to its terminal
                If the calling process is ignoring or holding SIGTTOU
                or if terminal output has not been inhibited
                        Return ACCESS-ALLOWED
                Else, if the calling process's group is orphaned
                        Return ACCESS-DENIED
                Else, send SIGTTOU to its process gro
            Process any signals pending for this process
  End Forever
```

Figure 1 - Job Control Line Discipline

session leader allocates the controlling terminal for its session, and is later inherited by that process's children. Traditional implementations store all a process's controlling terminal state information in that process's `user` area, making that information generally inaccessible unless the process is the currently executing process.

Normally, a session leader will allocate a controlling terminal before forking any children. However, if it reverses these actions, first forking children and then allocating a controlling terminal, a basic inconsistency arises between those children and the controlling terminal. The children will have inherited *session membership* from their parent, but will not inherit their parent's controlling terminal. Attempts to open `/dev/tty` by the parent will succeed; attempts by the children will fail. Yet as far as the controlling terminal is concerned, the process is in its session, and is subject to being sent signals.

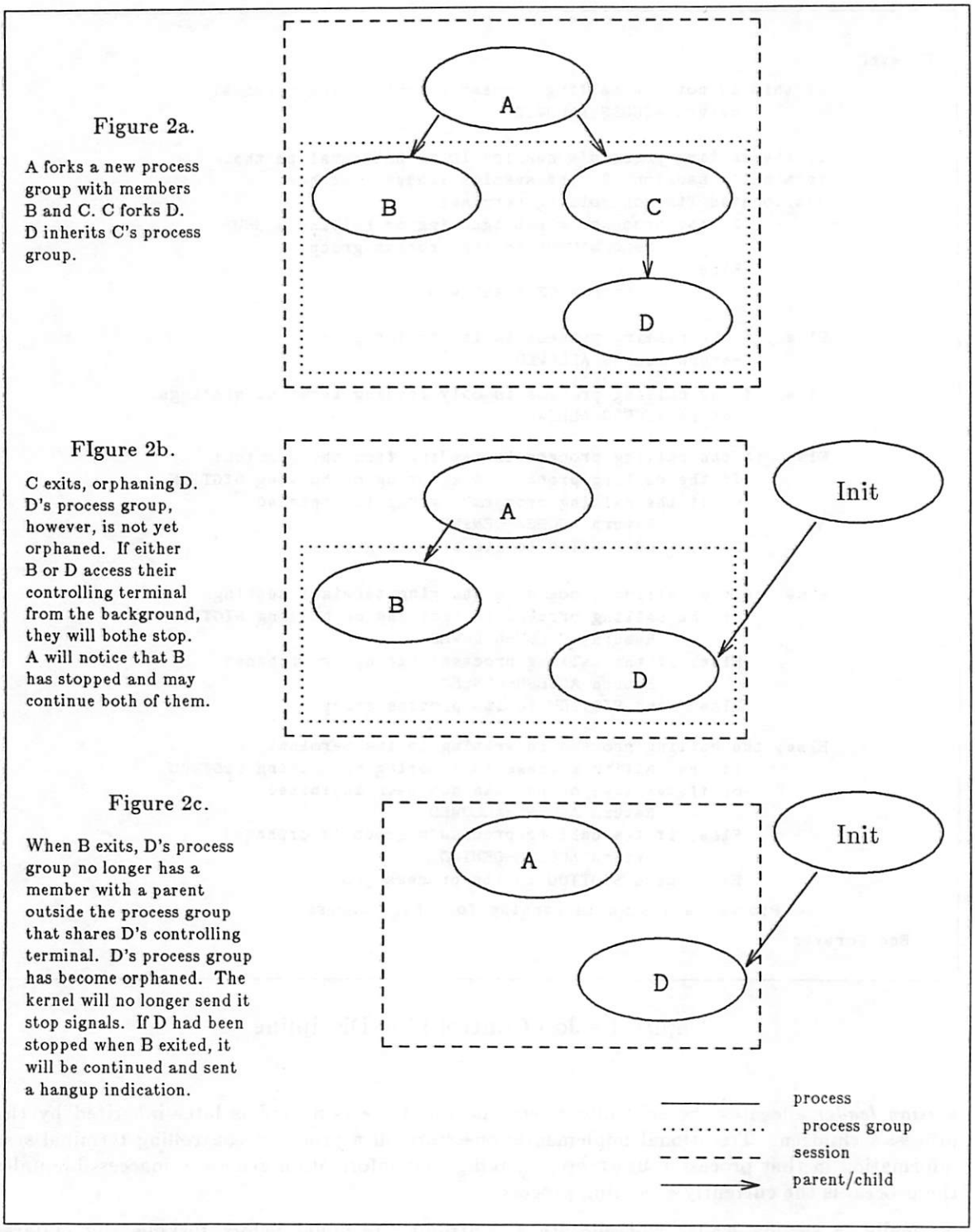


Figure 2 - Orphaned Process Groups

The problem is easily solved by reorganizing the process's data structures. The controlling terminal state information is moved into the `session` data structure, which is kept resident in kernel memory. There is only one copy for all processes in a session; processes inherit pointers to this structure, not copies of the information in the structure. Any changes to the contents of

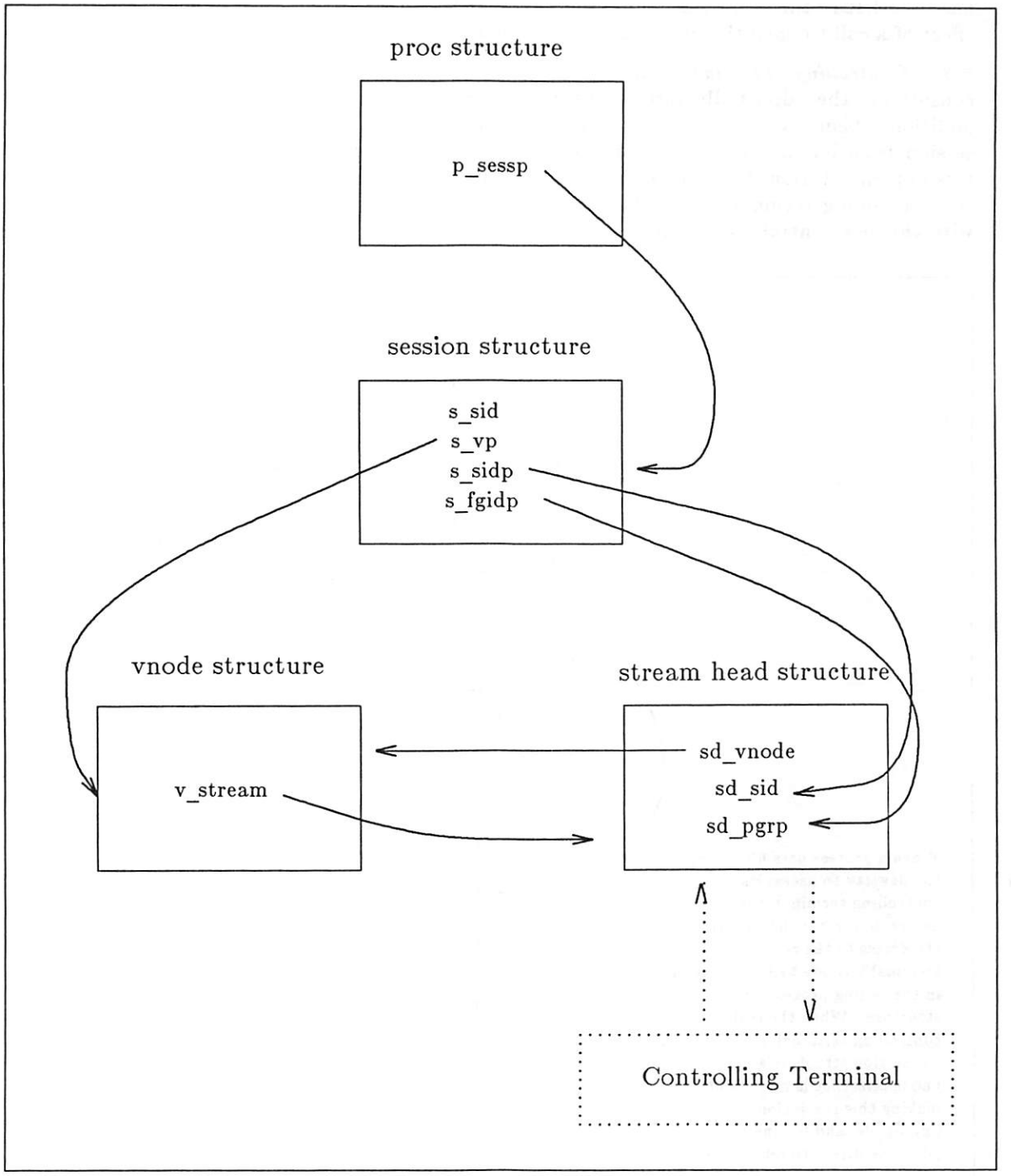


Figure 3 - The New Controlling Terminal Structure

the structure are therefore immediately made available to all processes in the session. Whenever a controlling terminal is allocated, it becomes the controlling terminal of all members of the session.

For backwards compatibility with existing terminal device drivers, the old `u_ttyp` field, typically modified during device opens to indicate that a controlling terminal has been allocated,

has been left in the user area. The kernel senses changes in the contents of this field as the side effect of a call to open the device driver and modifies the new data structures accordingly.

2.3.2 Controlling Terminal Reusability This reorganization might seem a radical change considering the admittedly rare occurrence of the inconsistencies it solves, were it not for the additional benefits it brings. For one, it removes one of the current obstacles that prevent a session from having multiple controlling terminals allocated to it during it's lifetime. Prior to this implementation, this was impossible, since it was impossible to find all existing references to the controlling terminal in the `user` areas of every member of the login session and update it with the new controlling terminal's state information. Now this update is trivial.

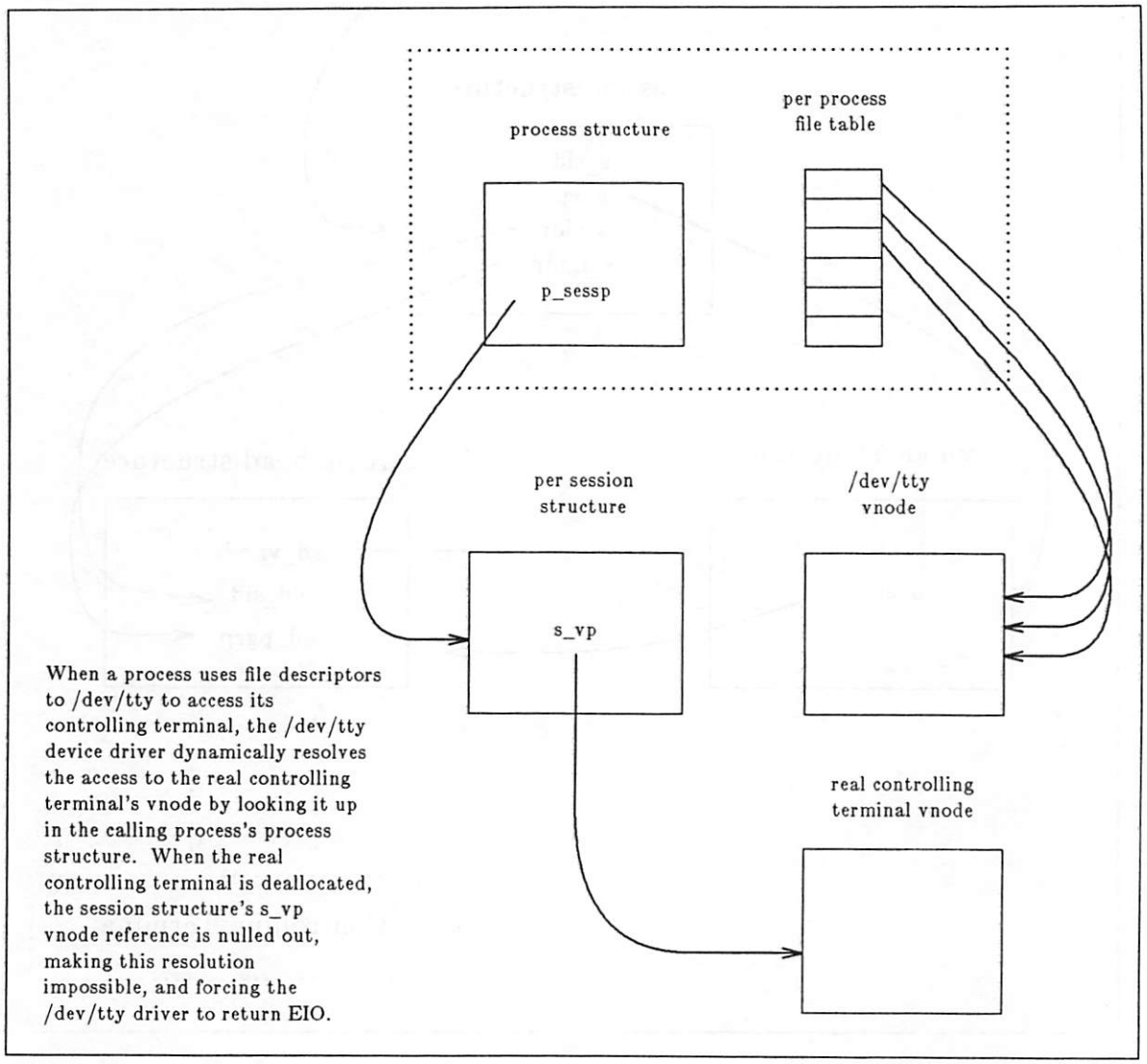


Figure 4 - the /dev/tty driver

The only additional obstacle is the lack of a reliable way to deallocate the controlling terminal. A controlling terminal is reliably deallocated when it's *session leader* exits, but this leaves the session without a process with the ability to allocate a new controlling terminal. Controlling terminal's are also deallocated when every outstanding file descriptor to the controlling terminal's device driver is closed. This, however, is anything but reliable; since file descriptors

to the controlling terminal are typically inherited by all offspring of the session leader, the session leader would either have to wait for all session members to exit, or force them closed by killing them. Worse yet, processes from other sessions (malicious or otherwise) can prevent the controlling terminal from being deallocated simply by successfully opening and holding open a file descriptor to it.

A visible symptom of this problem manifests itself with the `exec` builtin to the shell. `exec` allows a user to modify the shell's standard input, standard output and standard error; yet, if all are redirected away from the controlling terminal, it may or may not cause the current controlling terminal to be deallocated, depending on the existence of additional file descriptors to the shell's controlling terminal.

The principle of consistency suggests that the correct way to fix this problem is to keep a reference count of the number of times that the session leader opens the controlling terminal in the `session` data structure, and deallocate the controlling terminal when the session leader does the corresponding number of closes, regardless of any additional opens that may have been done by other processes in the system. This solution does not in general introduce incompatible behavior, since the session leader can at no time be guaranteed that either additional opens by other processes have been done; it merely makes the deallocation much more reliable.

2.3.3 Controlling Terminal Reconnection Often, a session prefers reliable reconnection to a controlling terminal that has lost carrier, not allocation of a new controlling terminal. The new architecture also supports a reliable policy for reconnection. When loss of carrier is detected, the controlling terminal will remain allocated. The session leader is sent a hangup indication (`SIGHUP`); in addition, if its process group was not in the foreground, the current foreground process group will be stopped (`SIGTSTP`) to temporarily prevent it from encountering any I/O errors on output until the session leader decides whether or not to reconnect to the terminal. Subsequent background process will be stopped as usual (`SIGTTIN` and `SIGTTOU`) whey they attempt to access the terminal; if they choose to ignore the stop signals while the terminal is disconnected, I/O errors will be generated.

Should the session leader exit at this point, it will start a chain reaction causing all process groups in its session to eventually be orphaned; the stopped process groups will be continued (`SIGCONT`) and sent hangup indications (`SIGHUP`). Alternatively, the session leader may reconnect to the terminal and reinitialize it creating a new file descriptor to it with a successful call to open after carrier has been reestablished. All existing file descriptors to the terminal will become usable again, and the new file descriptor may be discarded. Sending `SIGCONT` to the foreground process group will restore the session to the point it was before the disconnection.

2.3.4 Controlling Terminal Security Of overriding concern in the new session architecture is desire to create secure controlling terminals; processes in one session should not be able to access the controlling terminal of processes in another session unless they have gained permission to do so via standard file system permission checking mechanisms.

The root of this problem lies in the fact that open file descriptors are inherited by child processes. Since typically standard input, standard output and standard error are the most likely to be inherited from a process's parent, and since these are typically file descriptors to the controlling terminal, sessions typically have quite a few processes with file descriptors to the controlling terminal that the process has not explicitly opened. In order to maintain the maximum amount of backward compatibility, when a controlling terminal is deallocated by a session leader, only these inherited file descriptors should be invalidated; all others should be left alone.

The solution to this problem lies again in the new implementation of controlling terminals, as well as the standard implementation of the `/dev/tty` driver. When a process *read*()'s , *write*()'s, or *ioctl*()'s its controlling terminal via file descriptors created by opening `/dev/tty`, the device that is accessed is dynamically resolved by using the `vnode` reference stored in the process's controlling terminal state information. Since that information is now in the `session`

data structure and will be invalidated when the controlling terminal is deallocated, operations on file descriptors that were inherited by opening the `/dev/tty` driver will fail whenever the session looses its controlling terminal. The link from the `session` data structure's vnode pointer to the real vnode of the process's controlling terminal will be broken, so the kernel will not be able to resolve `/dev/tty's` generic vnode to the process's real controlling terminal. All operations will return an **EIO** error to the user.

2.3.5 Redirecting Controlling Terminals A final advantage to the controlling terminal reorganization is that it affords the opportunity to redirect existing file descriptors of session members to a newly allocated controlling terminal, transparently to those session members. This again, is serendipitously available because of the way the `/dev/tty` driver is implemented; since the device that is accessed is dynamically resolved and since that structure will be modified when a new controlling terminal is allocated to the session to reflect the `vnode` of the newly allocated controlling terminal, these file descriptors will dynamically resolve to the newly allocated device. Processes that inherit standard input, standard output and standard error file descriptors that were created by opening `/dev/tty` instead of those traditionally created when `getty` explicitly opens a terminal device, will have those file descriptors transparently redirected to all the controlling terminals for the lifetime of the session.

3. Future Work

SVR4's session management architecture provides the "hooks" for added value applications in two other areas.

It would be easy to implement *actively* reconnectable login sessions, where a session leader, after sensing disconnection, takes it on itself to establish connection with another controlling terminal. *Passively* reconnectable login sessions in which a session waits for a reconnection, requires a model similar to the one proposed in [4], where the traditional responsibilities of a `getty`-like process are divided into co-operating session manager processes that manages the creation of and re-connection to login shell and line manager processes that manage the allocation of terminals and authentication of users accessing the system.

In addition, the new model must cope the realities of the countless number of existing programs that have terminal dependencies embedded within them, typically via use of the `termcap` or `terminfo` databases. This problem is easily (albeit tediously) solved with the creation of pushable streams modules that translate from some generic set of terminal control strings to terminal specific control strings. By default, a user could specify their `TERM` environment variable to generate the new generic terminal control strings. The user would then push an appropriate translation module for the current terminal device allocated to their session.

4. Acknowledgements

Many of the ideas about the process group structure originated in conversations with Dave Lennert, David Korn and Larry Wehr in conjunction with AT&T's participation in the POSIX standardization effort. Countless refinements to that structure were suggested during the meetings of a POSIX job control working group that included Dave Lennert, David Korn, John Quarterman, Mike Karels and Richard Stallman.

5. References

[1] IEEE, "Portable Operating System for Computer Environments," P1003.1, 1988

[2] "Unix System V Release 3 STREAMS Programmer's Guide", 1986

[3] D. C. Lennert, "A System V Compatible Implementation of 4.2BSD Job Control," USENIX Conference Proceedings, Summer 1986.

[4] S. M. Bellovin, "The Session Tty Manager," USENIX Conference Proceedings, Summer 1988

```
close(0);
close(1);
close(2);
```

Free up descriptors 0-2.

```
open("/dev/mytty", O_RDWR);
```

Allocate controlling terminal.

```
SaveFd = fcntl(0, F_DUPFD, 4);
```

Save descriptor to real device above stdin, stdout and stderr. Controlling terminal will remain allocated as long as this descriptor remains open.

```
close(0);
```

Free up descriptor 0.

```
open("/dev/tty", O_RDWR);
fcntl(0, F_DUPFD, 1);
fcntl(0, F_DUPFD, 2);
```

Open pseudo controlling terminal driver as stdin and dup it to stdout and stderr. Children of this process will inherit descriptors to this device which will be automatically invalidated when the terminal is deallocated and automatically redirected when a new terminal is allocated.

```
fcntl (SaveFd, F_SETFD, FD_CLOEXEC);
```

Mark the descriptor to the real device as private.

Figure 5 - Programming Session Leaders

The Modix Kernel

Greg Snider
Hewlett-Packard Company
1501 Page Mill Road
Palo Alto, CA 94303

Jim Hays
Hewlett-Packard Company
19447 Pruneridge Ave.
Cupertino, CA 95014

ABSTRACT

The Unix kernel has evolved into a tightly-coupled, monolithic system that is expensive to maintain and difficult to configure. The Modix project pulled together a number of software engineering principles into a strategy for developing a modular, portable, highly configurable "parts catalog" for generating Unix kernels. A module hierarchy within the catalog was built using a dependency relation called *uses*. Modules were constrained to follow the object model [7], and cyclic *use* dependencies were not allowed; although adhering to the object model presented no problems in implementation, eliminating cyclic dependencies turned out to be surprisingly difficult. These structural goals had the side-effect of making deadlock avoidance in multiprocessor and realtime configurations easier and more localized than in some other approaches [1,6,13], and also made it possible to do rigorous testing. A prototype built to explore the ideas was partially completed.

1. Introduction

Most existing implementations of the Unix kernel are tightly coupled systems that are difficult to modify and extend. Their structure fails to meet the National Computer Security Center (NCSC) assurance requirements for trusted system classes B2 and higher [3]. New kernel engineers require a long learning period before they are effective due to the difficulties in extracting knowledge of the structure from the code (as has been pointed out, Unix "architecture is not manifest in its implementation but exists primarily in the minds of its designers" [12]). The lack of clean boundaries between subsystems can cause interference between kernel groups working in different functional areas. Given the lack of visible structure and the lack of structure enforcement mechanisms, there is little inducement for engineers to maintain abstractions that are, quite often, elusive; as a result, the coupling within the system tends to increase with time. The structure also makes it difficult to configure feature sets, forcing nearly all features, whether needed or not, into one monolithic product.

2. Problem

The problem we faced was that of simultaneously satisfying a number of different, and sometimes conflicting, requirements within a single Unix kernel. Specifically, we wanted to support:

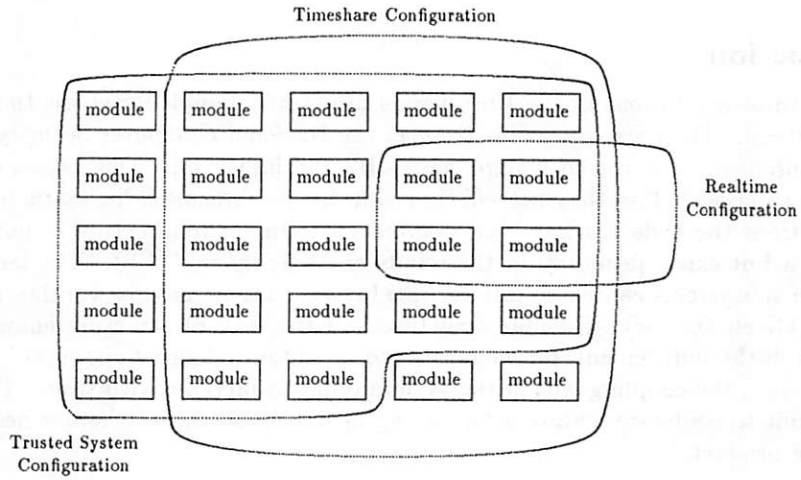
- B3 class security.
- Symmetric multiprocessing.
- Realtime applications.

- Small configurations.
- Portability.

This is quite difficult. For example, an embedded, realtime system may not have to be secure, and may not want the additional run-time overhead (such as auditing) needed in a trusted system [5]; conversely, a B3 trusted system might have a difficult time supplying the fine-grained timing and scheduling requirements of realtime applications without introducing covert channels with unacceptable bandwidth. Trusted systems are also harder to make major modifications to since assurance requirements dictate that changes to protection critical code be carefully reviewed. Of course, one could build a separate, specialized kernel for each class of users — a realtime kernel, a secure kernel, etc — but the cost of supporting multiple Unix kernels was judged unacceptable. The challenge, then, was to come up with an approach that could not only satisfy different market segments with different needs, but that had maintenance costs comparable to those of a single, general-purpose kernel.

3. Objective: Building Block Kernel

The idea was to create a Unix "parts catalog" consisting of relatively independent modules that could be linked together in different ways to create different types of kernels, much in the way that a traditional Unix I/O system can be configured to support different sets of peripherals. The modules would be thought of as building blocks: different kernels formed from the catalog would really be nothing more than different configurations of the blocks. Some of the blocks might be alternate implementations of a particular service - for example, there might be two or more scheduler modules, each geared to a particular environment (generic timeshare scheduler, realtime deadline scheduler, fair-share scheduler, etc.). But to make this scheme practical, it was important that a large proportion of the modules, say 80%, be usable in more than one configuration.



To simplify support and testing, the number of kernels configured from the catalog would be kept to a minimum. In the ideal case, only one kernel configuration — a configuration that satisfied all of the needs for security, realtime, multiprocessing, small systems, etc. — would be supported, but we doubted that a single configuration could supply acceptable performance and functionality for all the different market needs we were trying to address.

4. Constraints

For this building block approach to work, the structure of the parts catalog had to:

1. satisfy all functional requirements, *and*
2. localize functionality that is not required for all configurations.

The different functional requirements place constraints on acceptable kernel structure. The security requirements are especially constraining, but even the multiprocessing requirement limits the design freedom in producing a kernel that is also maintainable. Functions that are not common to all configurations need to be highly localized within one module or set of modules to make configuration practical. For example, if security extensions were distributed uniformly throughout the modules, configuring a non-secure system would be very difficult. Some examples of functions that may not be desirable in all configurations are:

- trusted system access checks.
- deadline scheduling.
- remote file systems (e.g. rfs or nfs).
- multiprocessor scheduling.

In addition, we also wanted to meet the NCSC's "Orange Book" requirements for B3 class trusted systems [3]. The B3 evaluation class imposes a number of requirements, but this paper will address only two of them, namely that the trusted part of the system:

- "[be] internally structured into well-defined largely independent modules."
- "incorporate significant use of layering, abstraction, and data hiding."

Note that these are not sufficient; there are other system architecture requirements that have to be addressed as well, but they are beyond the scope of this paper.

One might think that, with some work, Unix could be coerced into meeting these requirements, but the task is not trivial. An NCSC study of Unix concluded that generic Unix does not meet the B2 (and, as a consequence, B3) modularity requirements, and that "bringing an existing system to the B2 level is likely to be at least as difficult as building a brand new system." [12]. The study made a number of suggestions for developing a design strategy that addressed modularity. Among these were:

- The elimination of globals whenever possible.
- Making the system's modularity and structure evident and consistent.
- Documenting coding practices, packaging rules, data structure design rules, and the module hierarchy.

5. Prototype

To investigate how these objectives might be realized, we built a small prototype. Due to limited time and staffing, our emphasis was on structure and configurability rather than functionality. As a result, our prototype addressed only the general areas of process management, memory management, and the file system — we ignored I/O, networking, and distributed systems issues. The remaining sections describe the approach we took and what we learned.

6. Structure

Much of the complexity of Unix derives from the high degree of coupling or connectivity within the kernel. Our hypothesis was that limiting and constraining the connectivity of the system would reduce its complexity. Although intuitively we felt we knew what connectivity was, we first sought out a more formal definition.

6.1 Dependency and Use Relations

The *dependency* relation provides a means for describing and evaluating the connectivity of modules in a system. The definition that we adopted for this relation is an extension of Freiertag's and Janson's dependency relation [8].

A module, call it *A*, is said to *depend on* another module, *B*, if the correct operation of *A* cannot be verified without first verifying the correct operation of *B*. There are three different classes of *dependency* relationships:

1. *service dependencies:* *A* invokes a service in *B* and uses results or side-effects of that service. The service may be invoked through a function call, message, signal (e.g. a semaphore operation), or through hardware, such as via a trap.

2. *data dependencies:* *A* shares a data structure with *B* and relies upon *B* to maintain the integrity of that structure.

3. *environmental dependencies:* *A* does not directly invoke *B* and does not share a data structure with *B* but nevertheless depends upon *B*'s correct functioning.

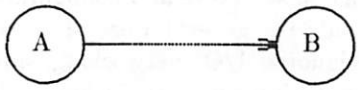
The definition of *service dependency* might suggest that an invocation of a service within a module always implies a *dependency* of the invoker on that module, but there are some invocations that do not imply a dependency. If *A's* specification merely requires that it invoke *B* under certain conditions, but does not depend on the results of *B's* execution, then *A* does not *"depend on"* *B*. Or put another way, if *B*'s entry point were to be replaced by a null routine and *A* still performed correctly according to its specification, then *A* does not depend on *B*; rather, *A* is merely notifying *B* of an event that it is presumably interested in.

The second class, *data dependencies*, shows how global variables can increase module connectivity. Modules that share a writable data structure become interconnected and mutually dependent on each other.

The third class, *environmental dependencies,* is slightly more subtle. One example is the dependency of most of the system on the interrupt handling subsystem. Although this is not generally called directly from the kernel, much of the kernel depends on its correct operation (e.g. properly restoring processor state at the end of an interrupt service routine) in order for the kernel to fulfill its specifications.

In practice, we did not find that environmental dependencies presented many structural problems. So instead of using the dependency relation for characterizing connectivity, we used a slightly different relation called *uses* (not identical to the *use* relation advocated by Parnas [10]). A module is said to *use* another module if it has a service dependency or data dependency on it.

A *use* relation between two modules is represented graphically with an arrow drawn from the "user" module to the module it depends on. For instance, the following graph represents two modules, *A* and *B*, where *A uses B*:



Modix used the *use* relation to define the module hierarchy, as will be seen later.

6.2 What's a Module?

Eliminating shared variables eliminates data dependencies altogether, so we adopted the elimination of shared variables as a goal. The object model is a design methodology that allowed us to achieve that goal [7].

A system designed using the object model consists of a set of passive components called *objects.* An object represents an abstract resource, and may be characterized by a set of invariant

properties that describe the resource's behavior. These properties are preserved by the object itself in order to maintain a coherent model of the resource and cannot be subverted by an agent outside the object; this is accomplished by limiting access to the object to a set of interface *operations*. The number of operations on any one object is fairly small, typically a dozen or less. Internal state information cannot be directly accessed from outside. The behavior of the object can thus be defined in terms of stimulus/response, where the stimuli are those operations that make up the object's interface.

Objects may be thought of as being created from a template known as a *class*. The class template defines the data structure for representing the object, along with the operations on that object. The class typically does not actually allocate any space for an object's data structures — it merely contains the pattern for them, along with the code for implementing the object's operations. Using the template to create an object can be referred to as creating an *instance* of that class; many different instances (objects) for a given class can be created. Each instance is an independent entity: each has its own private data structures, but shares the class's code that implements the operations. Instances may also share *class variables* — variables that instances of the class have equal access to, but that are generally inaccessible from outside.

So, what is the relationship between a *module* on the one hand, and *objects* and *classes* on the other? We defined a module to be a class along with zero or more instances of that class. This resulted in three interesting cases:

0 instances: The module was a template for creating objects. Other modules could use this to create their own local instances of the class.

1 instance: The module was an object. If the class was made public, rather than private to the module, it could also be used as a template for creating objects.

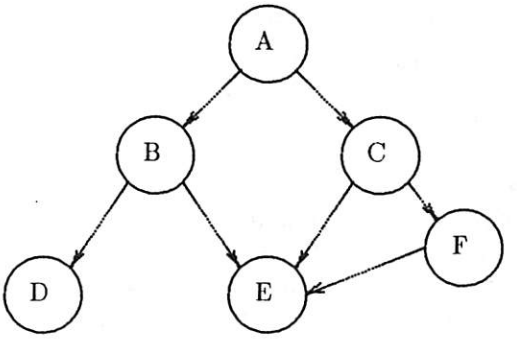
>1 instance: The module was a pool of objects. This invariably meant that class variables were required to manage the pool.

Our constraints on classes were quite strict. We allowed no access to data members from outside of the class. By carefully designing our abstractions to meet these constraints, we achieved data hiding (and algorithm hiding as well) and eliminated data dependencies from the system.

6.3 Cyclic Dependencies

Although the object model is a useful tool for eliminating data dependencies, it does not address the topology of the module interconnections. The topology of the interconnections can greatly affect the complexity of the system as a whole.

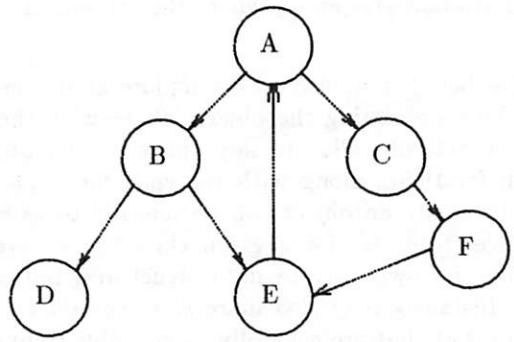
Consider a simple system made of 6 modules, *A, B, C, D, E,* and *F,* with the following *use* relations:



In this system, testing and verification can be done in a straightforward manner. First, *D* and *E* can each be verified in isolation since they have no dependencies on any other part of the system. Next, *B* and *F* can be verified since all of the modules that they depend on, namely D and E,

have already been verified. Then C can be verified, and finally A. This verification is simple since its focus is always local at every step.

Consider the consequences of removing the dependency of C on E and adding the dependency of E on A, though:



The total number of connections in the system remains the same, but the system's personality is changed radically. Here, D can be verified since it has no dependencies on any other part of the system. After D, though, there are no modules left that are independent of unverified parts of the system. The difference between the two graphs is that the first is acyclic, while the second has cycles of dependencies that entangle five of the six modules. Verifying the modules in a cycle is much more difficult because there is no place to start — *the entire cycle can only be verified as a whole.* Not only is verification more difficult, understanding the system is more difficult. Such a system requires a more global perspective, especially when dependency cycles are large and encompass a large number of modules.

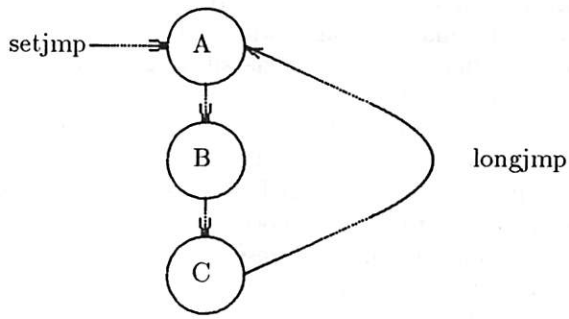
Cyclic dependencies not only complicate testing and verification, they make configurability harder to achieve. In the first system eleven different configurations are possible: *D, E, DE, EF, BDE, FDE, CEF, BDEF, CDEF, BCDEF,* and *ABCDEF.* In the second system, cyclic dependencies reduce the number of possible configurations to only two — *D,* and *ABCDEF* — thereby limiting configuration flexibility.

Subtle service dependencies can also introduce cycles in a system if one is not careful. For example, a cycle could occur in a virtual memory system if virtual memory data structures are actually allocated in virtual memory (e.g. pageable page tables). The cycle here results from the fact that part of the system actually resides within the very abstraction it is trying to construct. Another example is the interaction between a file system and a virtual memory system that supports memory-mapped files. The file system might like to exist on top of (depend on) virtual memory, but if a process page faults while reading a memory mapped file, the virtual memory system may need to depend on the file system for resolving the fault. Although avoiding this type of circularity is difficult, particularly in virtual memory systems [9], it can be done [8].

For these reasons, dependencies between modules in Modix were restricted to those that could be represented by a directed, acyclic graph. Put another way, dependencies between modules in the system had to be partially ordered.

6.4 Exception handling

One particularly insidious form of intermodule dependency results from the *setjmp() / longjmp()* mechanism that has traditionally been used in Unix. Consider the invocation sequence in the following figure where object A calls *setjmp()* and then invokes an operation in object B, which in turn invokes an operation in object C. (Note: the arrows in the following figure denote flow of execution, not dependency relationships).



If C can then *longjmp* back to A, then B has an added constraint on its implementation that may not have existed otherwise. For example, in this scenario B can only call C if B's data structures are in a consistent state; calling under any other conditions could destroy the coherency of the object. The situation becomes worse as the number of objects between the object invoking the *setjmp* and the object invoking the *longjmp* increases — all such modules then have hidden dependencies that might easily be missed.

As an example, consider a set of objects, O_1, O_2, ..., O_n, linked in a long chain of invocations (i.e. an operation in O_1 invokes an operation in O_2, which invokes an operation in O_3, etc.) where the last operation in the chain (in object O_n) *longjmp*'s back to the first. Furthermore, assume that each object is coded to be reentrant and uses semaphores to synchronize its operations' use of its internal data structures (actually this assumption is unnecessary, but helps make the problem easier to see). An operation in O_2 might naturally want to lock a local semaphore and invoke an operation in O_3, which might then lock a semaphore and invoke an operation in O_4, and so on down the chain. When the flow of execution reaches O_n, objects O_2 through O_{n-1} are all in tentative states — execution has been suspended within each of them in a critical section with a semaphore locked. The longjmp out of O_n leaves the intermediate objects in this tentative state and precludes any simple means of completing the suspended critical sections and unlocking the locked semaphores.

One possible solution here is to constrain the implementations of O_2 through O_n so that such inconsistent states cannot occur. The constraint might be worded like this: while inside a critical section, an object may not invoke an operation in another object if that operation could potentially fail to return control due to a *longjmp* somewhere in the invocation chain. Compliance to such a constraint, however, would be difficult to verify and could easily be overlooked during implementation and maintenance.

It appeared that language support was needed to handle this situation in a "clean" and efficient way. Since Bjarne Stroustrup, the designer of C++, has hinted that C++ will eventually provide such support [15], we took the path of least resistance: setjmp's and longjmp's were disallowed. Instead, exceptions were explicitly returned back up the invocation chain, thus allowing each module to do whatever cleanup was needed. A more elegant solution was deferred until language support was available.

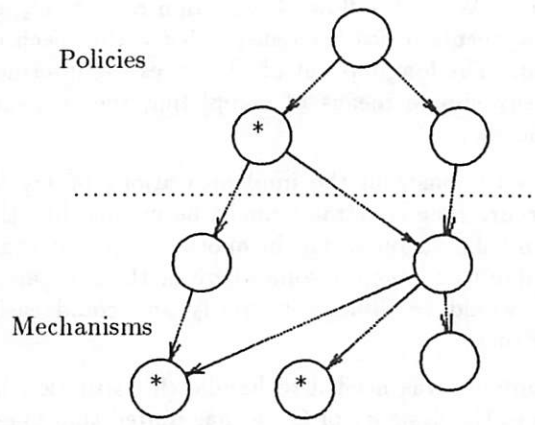
6.5 Layering

The strategy for organizing modules into *layers* partially resulted from B3 requirements not covered in this paper. In addition, we tried for a layering scheme where design decisions about system policies tended to reside in higher layers, while mechanisms for implementing those policies tended to reside in lower layers. We were also interested in localizing machine dependent modules, perhaps by using the hardware-dependent/hardware-independent (HI/HD) layering (or virtual machine approach) as was done in the SunOS and Mach virtual memory systems [4,16]. We found, however that policy/mechanism layering and HI/HD layering are not always compatible. The reason: some design decisions which one would be tempted to label as policies rather than mechanisms can be machine dependent.

One example of this incompatibility arose during the design of the virtual memory subsystem. The code for creating virtually mapped pages within a process's virtual address space is a fairly low-level, machine dependent function that we classified as a mechanism. At a higher level, contiguous spans of virtual pages can be collected together to form segments. Above that lies the problem of deciding how segments should be laid out in an address space, for example, where the text, stack, and data segments go, where segments for implementing shared memory get placed, etc. However, this layout problem, which to us fell more into the category of policy, was machine dependent. For instance, the location of memory-mapped I/O registers (if any), the location of kernel code and data (which consumes user address space in some systems), and address space segmentation (e.g. if ring protection mechanisms are tied to addressing) are machine dependencies which affect the segment layout policy.

One might try to solve this by arguing that anything that is machine dependent is a mechanism and then coercing the system structure to support that, either by moving such modules "lower", or by moving the boundary of the machine dependent layer "higher". Attempting to move such modules down, though, created conflicts with our goal of not allowing cyclic dependencies. And moving the machine dependent boundary up only made the machine dependent layer larger than it really had to be since the *use* relations would then force machine *independent* modules into the machine dependent layer.

Our solution was to abandon HI/HD layering altogether, replacing it with module "tags" that would mark a module as being machine dependent or not. Since the module interfaces were well defined and data connectivity was not allowed, replacements for hardware dependent modules for different machines could be written without impacting the rest of the system, thus achieving the real goal of HI/HD layering.



* => hardware-dependent

Of course, the boundaries between mechanisms and policy and between machine dependent and independent modules can be somewhat fuzzy. What may be hardware independent on one set of architectures may still require changes to run on some other architecture outside of that set. And it is sometimes worthwhile to modify "hardware independent" code when performance can be improved on a specific architecture. Still, it's a useful way of pointing out where changes are likely to be needed when porting the system to a new machine.

6.6 Implementing Modules

C++ was our first choice for an implementation language since it could support and enforce the modularity we were trying to achieve: we were afraid that the modularity of a system like this would erode too quickly in C. C++ would also let us cut and paste existing C code to help implement our prototype. However, a C++ preprocessor (let alone a compiler) was not available to us when we started, so we instead adopted a style of writing in C that would make later

conversion to C++ possible (although some C++ features, such as inheritance, turned out to be difficult to emulate).

A module consisting only of C code (no assembler) was represented by a (.h file, .c file) pair. For example, a module representing a class named *Foo* would be implemented in files *foo.h* and *foo.c*. The .h file contained the class declaration, declarations of the class operations, and "inline" operations (which were simulated using *#define* macros). The foo.h file might look like:

```
#define class typedef struct

class {
/* private: */
        int  member_1;
        Bar  member_2;
} Foo;

extern foo_op1();
extern foo_op2();
#define foo_op3(x,y) {x->member_1 = 2*y}
```

Class variables, optional instances of the class, and the bodies of the non-"inline" operations, resided in the .c file.

Operation invocation, which might be expressed in C++ as

```
foo_instance.op(arg1, arg2);
```

was instead written as

```
foo_op(&foo_instance, arg1, arg2);
```

Modules containing multiple instances of a class (handy for managing fixed-size resource pools) generally required allocation and deallocation routines to obtain instances. In that case, the allocation routines were regarded as operations on the class rather than on a specific instance of that class, e.g.:

```
foo_instance = foo_allocate();
foo_op(foo_instance, arg1, arg2);
foo_deallocate(foo_instance);
```

Modules consisting of pure assembler (there weren't many) were done much the same way. A .h file, consisting generally of assembler macros, was combined with a .s file to create a module. Occasionally a low-level module would require both C and assembler. To handle this, the .h and .c files of C code were combined with the assembler files to form the module. To show that the individual files were logically one, their root names were the same, except that assembler files were required to have a trailing underscore before the .h and .s suffixes. Thus, a module *Foo* could be implemented as the file tuple (foo.h, foo.c, foo_.h, foo_.s).

7. Synchronization

Since we wanted to support multiprocessing and realtime, we needed to develop appropriate synchronization mechanisms for increasing the concurrency of the kernel. The traditional *sleep()*, *wakeup()*, and *spl()* mechanisms were not used because of the race conditions they present in a shared-memory multiprocessor environment. Much of our strategy here was taken from experience gained from an earlier experiment in semaphoring the HP-UX kernel.

7.1 Mechanisms

Early on we noticed that synchronization problems tend to fall into two classes. The first class, *mutual exclusion*, occurs when it is necessary to ensure that critical sections with respect to kernel data structures are only executed by one process at a time. This can be looked at as means of protecting a resource that can be used over and over again in the system, but only by one process at a time. The second class, *producer-consumer synchronization*, occurs when one process, the consumer, must wait for another process to "produce" a resource that it needs before it can continue execution. The resource produced need not be concrete — it could, for example, be a clock tick or an interrupt from a device. The resource itself is generally not reused; it is either relayed out of the system or discarded. (More complex types of synchronization problems can be modeled as elaborations of one or the other of these classes. Barrier synchronization [14], for example, can be modeled as a set of producer-consumer relationships among a number of cooperating processes.) Although a single mechanism, such as semaphores, can be used for both classes of problems, we found it advantageous to use different mechanisms for each class.

Mutual exclusion was provided by two different lock abstractions: *spinlocks*, and *mutexes*. Either can be used to guard critical sections by requiring a process to acquire one of them upon entry to the section and to release it at exit. If this rule was followed, the implementation of these abstractions guaranteed that only one process at a time could execute within any critical section protected by a given lock. Both locks types have in common the property of ownership; a process *owns* a lock when it acquires it, and loses ownership when it releases it. Only one process may own any particular lock at any given time. One result is that acquisition and release of either type of lock is usually textually linked: a sequence of code that acquires a lock at one point must at a later point in the sequence release that lock (an exception to this occurs when a serially reusable resource is locked with a mutex and passed to another process — in that case, mutex ownership is transferred as well and lock and unlock operations may not be textually linked).

Spinlocks were used to protect data structures accessed from interrupt service routines; in our implementation, spinlocks also disabled processor interrupts to prevent certain deadlocks involving interrupt service routines that could otherwise occur. Since spinlocks require very little overhead, they were also the preferred mechanism for protecting data structures when the following constraints could be met:

- A process could never block, voluntarily give up the CPU, or generate a trap that could cause the executing process to be suspended while the lock was owned.
- A process could never hold the spinlock longer than 500 μsec on our target machine (necessary to meet our realtime preemption-delay requirements).

Mutexes were used to protect data structures that could not be protected by spinlocks. Conceptually, mutexes are very similar to binary semaphores, but with the additional ownership property mentioned previously (which was a nice property to have during debugging). They differ from spinlocks in that they do not cause interrupts to be disabled while held, and cause the executing CPU to switch to another process if the desired resource is not immediately available. Processes blocked on mutexes were sorted by priority; whenever a mutex was released, the highest priority blocked process was given ownership of the mutex and set runnable.

Producer/consumer synchronization was provided by *event counts* and *semaphores*. We found semaphores to be surprisingly awkward at expressing some types of producer-consumer relationships, often requiring the use of additional state variables and spinlocks to avoid races. Event counts were often an easier and more "natural" way of expressing such relationships, but as J. Eric Roskos has pointed out, they have the disadvantage of first-in / first-out activation of processes blocked on the event count, when a priority-based activation order was desired for realtime [11]. Semaphores were thus used where this was a problem, but event counts were quite handy when it was known that there could be only one consumer (e.g. synchronous I/O).

7.2 Deadlock Avoidance

Deadlock can be avoided by assigning a "deadlock ordering" to all of the spinlocks and mutexes in the system and allowing a lock to be acquired by a process only if that process owned no other locks of the same or higher order. This avoids the classic binary deadly embrace where one process tries to acquire resources A and then B, while another process running concurrently tries to acquire B and then A. In semaphoring Ultrix, Hamilton and Conde actually assigned a formal number to each mutex and checked that mutexes were always acquired in increasing "deadlock order" during debugging [6]. We had done the same thing in a previous investigation of fine-grained semaphoring of HP-UX for realtime. Although this worked quite well, it is a somewhat "global" approach to the problem, since deadlock order values must be assigned from a system-wide scale. This was not consistent with with our goal of having a loosely coupled system.

It turns out that deadlock ordering did not have to be explicitly specified in Modix since it was actually implied by the partial ordering of the modules: in *use* calls, control could only flow downwards in the module use graph, and therefore mutex and spinlock acquisition at the module level was automatically partially ordered. Formal ordering of locks was only necessary within modules, when a module had two or more private locks that had to be acquired simultaneously. The only problem came from non-use upcalls — one had to be careful to release all locks held before making such a call.

Within a module, though, there were often cases where a simple ordering was insufficient. Either the algorithms required that two different resources be locked in different orders in different situations, or they required that two objects of the same class be locked before performing some operation on them. How to establish an ordering in that case? We used three different tactics:

1. Sometimes the algorithm itself imposed, or could be changed to impose, the ordering. Consider, for example, parsing a pathname in namei(): this involves a descent of the file system name space, iteratively locking a directory inode, and then holding that lock while locking the inode for the next component of the name. Ignoring ascents of the name tree and circularities due to links, the parsing traversal is deadlock free since the hierarchical file name space establishes a partial ordering on inode locking.

2. Other times, though, the algorithm could not easily be changed to guarantee such an ordering. An example of this occurs in the BSD buffer cache where brealloc() needs to lock two buffers at the same time, but with no simple way to establish an ordering on the buffers. A solution here is to add a special "contention breaker" mutex that must be acquired when two objects of the same class need to be locked [2]. To avoid deadlock, the contention breaker must be acquired before attempting to lock either of the objects, then the two objects can be unconditionally acquired and the contention breaker released. (The contention breaker does not have to be acquired when locking only one object in the class.) Since this protocol guarantees only one process can be attempting to lock two objects at the same time, deadlock is avoided.

3. Still other times, though, an algorithm change or contention breaker could not solve the ordering problem. The solution, which has been used in a number of systems, is conditional mutex acquisition. Bach and Burroff have written a very clear explanation of this tactic [1].

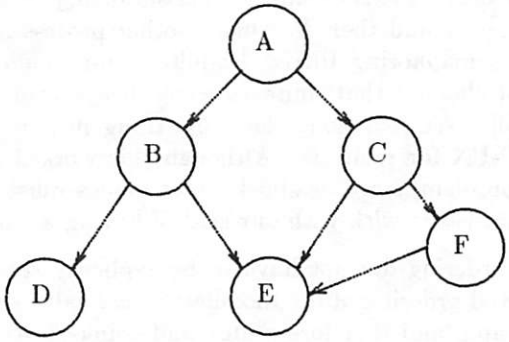
Thus, we found that our structural goals made synchronization easier for two reasons:

1. Critical sections were localized. Any given lock was visible only within a single module, and critical sections protected by the lock always started and ended within that module.
2. Deadlock avoidance was a natural outcome; the locking protocol came cheaply.

8. Configuration

The lack of cyclic dependencies allowed for the creation of a large set of "subkernels", each of which could serve as the base for a software system. In fact, it was possible to point at one or

more modules in the system catalog and build a subkernel based on those modules as the "roots." The subkernel would consist of the selected modules and all other modules that they recursively *used*. Consider, for instance, the simple system shown previously:



In this system, all eleven of the possible subkernel configurations (D, E, DE, etc.) could be built.

8.1 Dependency Specification

This configuration subsetting was made possible by embedding knowledge of the system structure into the system generation process itself. This was approximated in the first pass by mapping the partial ordering of modules in the system onto a total ordering; this allowed the system generator program to know which modules to link together to create the desired subkernel. However, we planned to replace this by putting the partial ordering into the code itself. The mechanism was a new preprocessor statement called #uses that replaced #include statements in the source code. The statement pretty much describes what it does: it specifies the *use* relationships among the kernel modules. The #uses statement was restricted to .h files; .c files contained no #uses or #includes. In the last example, the #uses statements in the .h files would look like this:

A.h	B.h	C.h	D.h	E.h	F.h
`#uses B`	`#uses D`	`#uses E`			`#uses E`
`#uses C`	`#uses E`	`#uses F`			

Thus, in compiling and linking a subkernel, no special configuration files are needed. The #uses statements imply which .h files are needed for compilation, and which modules to link together to create the desired subsystem. The #uses statement also documents the module hierarchy; given the source code, it would be straightforward to create a program that would draw a graph of the entire system's structure. It also makes it possible to automatically check the system for cycles in the *use* relation.

Non-*use* upcalls — non-dependent calls made upward through the calling hierarchy — did create a few problems for compilation and linking, though.

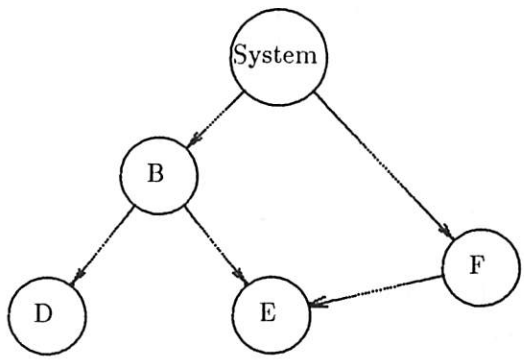
First, the #uses mechanism provided no way of declaring such upcall functions. For example, if module D made a non-*use* call into module B, the compiler would be unable to obtain any type information about the called function when compiling D. Since non-*use* calls cannot return information by definition, though, the type of any such function in B could only be (void). Hence, upcalls from within a module could simply be declared as void functions in that module. This would not have worked well in C++, though, since the compiler would have been unable to do type checking of the arguments.

Second, it was possible to create configurations where the bodies of such functions didn't even exist. If the subkernel consisted only of module D, which made an upcall to a function in module B, the code for the function in B would not be compiled and linked into the system. However, since the relationship was non-*use*, such functions could be replaced by null functions (e.g. `(void) upward_func(){ }`) in those configurations without disturbing the system (if that were not true, the relation could not have been non-*use*).

Third, upcalls occasionally created object naming problems. An upcall to a single unique object known at link time, or an upcall to a class operation presented no problems. However, there were a few cases where an object, *A*, would invoke a service in module *B* that could not return a result immediately but would have to asynchronously notify A of the result at some time in the future. In this case, *B* needed some sort of handle on *A* so that when results were available, *B* could send them to the proper object. The obvious solution was to have *A* pass a pointer to itself as one of the parameters to the service in *B*. But since *B* was compiled without *A*'s class declaration (because of #uses), such a declaration was not possible (in C). The problem was short-circuited by declaring such parameters as "generic" pointers. This allowed *A* to pass a pointer to itself cast as a generic pointer to keep the compiler happy.

8.2 The 'System' Class

The collection of modules that made up a subkernel also created another abstraction — that of the system itself. This abstraction was represented by a class called *System*, with exactly one instance. The *System* class was not of fixed composition, since the number of different systems that could be configured was large, but rather was built dynamically by the system generation process when the system composition was specified. Suppose, for example, that modules B and F in the last figure were specified as "roots" of a desired subkernel. In building such a subkernel, the generation process would deduce that modules D and E were also needed by the #uses statements, and tie the whole system together with the *System* class at the root of the hierarchy:



The *system* object was responsible for initializing the system and for giving the system direction when initialization was complete. It could be thought of as the "application" that the specific configuration was designed for. It had two operations: *system_init()*, which was built automatically by the system generation process, and *system_main()*, which was written by the configurer for the desired configuration.

(Actually the system class is just a special case of a *configuration dependent* class — a class whose composition and operations are a function of the system configuration. For example, the *process* class in our system was not of fixed composition but depended on the modules included in a specific configuration.)

8.3 Initialization

Every module was required to have an initialization entry point. For example, module Foo was required to have entry foo_init(). This entry point would typically initialize class variables and might, perhaps, create private instances of the module's class.

Given the root modules for the system being built, the system generation process would automatically create the code for *system_init()* in the *System* class. This function would be built to initialize each of the modules in the system in a "bottom-up" manner — at entry to an init() routine, a module was guaranteed that any and all modules it *used* were already initialized. After all modules were initialized, *system_init()* would then call *system_main()*. For example, the previous configuration would cause the system generator to create the following code for *system_init()*:

```
system_init() {
        D_init();
        E_init();
        B_init();
        F_init();
        system_main();
}
```

At power-up, control was first passed to a small piece of assembly language code at location *init*. This assembly code did the minimal amount of work necessary to create an environment for C (such as creation of a stack) and then transferred control to the *system_init()* entry in the system object.

8.4 Testing

The configurability of the system made it easy to rigorously test each module. Each module was required to have a test entry (enclosed by "#ifdef TEST ... #endif") that could do extensive black-box and white-box testing. To test a module, a subkernel was built with that module as the root. The module would be compiled with the test code included; all other modules would be compiled without their test code. After system initialization, control could then be transferred to the root module's test code to test the module in a very simple environment, exercising code paths and corner cases difficult to reach or create in the environment of an entire system. This proved to be very helpful in bringing up the prototype, and the small size of many of the subkernels allowed us to test and debug the lowest levels of the system on a CPU simulator rather than on the target machine.

9. Summary

The Modix project adopted the following strategies for creating a configurable parts catalog for generating Unix kernels:

- Localization of machine dependencies.
- Localization of optional features.
- Alternate module implementations for different applications or different architectures.
- Object model — to provide data hiding and eliminate data dependencies.
- *Use* relation to establish module dependencies.
- Acyclic system structure.
- Separate synchronization primitives for mutual exclusion and producer-consumer synchronization.
- Global deadlock avoidance leveraged from acyclic structure.
- Strategies for local deadlock avoidance (algorithm exploitation, contention breakers, conditional mutex acquisition).
- System structure expressed directly in code for documentation and system generation.
- Automatic configuration-dependent initialization.
- Arbitrary subkernel configuration — to create kernels for different applications and to allow for rigorous testing.

10. Conclusions

For various reasons, work on Modix was suspended after approximately nine months of work, before the prototype was completed. High level designs were completed for the virtual memory system, most of process management, and a small part of the file system. Most of the virtual memory system, trap and interrupt handling, and low level process management were implemented.

As a result, the experiment was only partially successful. Many of the advantages of the approach, such as testability and auto-initialization, were discovered only as a result of trying out the ideas of modularity and structure. One of our central objectives, though, was to determine whether it was possible to achieve acceptable performance with such tight structural constraints — we had conjectured that it was possible to design a system like this without degrading performance by more than, say, 5%, but this we were not able to show.

During design, the object model did not cause us many problems in decomposing the system. Although it may not be optimal for all problems, the object model did seem to be a very good match to the problem we were addressing, namely configurable kernel design.

Eliminating cyclic dependencies, though, consumed a large part of the design time, often forcing us to rethink and change our abstractions. It turned out to be surprisingly hard to do in some cases. In light of this experience, we do not know if it is always feasible to remove such dependencies; although a general technique for doing this exists (Janson's "sandwiching" approach [8]), it does not always render a solution as elegant as obtained by ad-hoc reformulation of the abstractions.

We were also unable to investigate how maintainable the structure would be under the time pressures of maintenance and enhancement requests. Our hypothesis was that the structure of this system would degrade more slowly than a conventionally structured Unix kernel since, at the least, the structure would more visible to engineers making alterations, and also since tools could be built to detect some violations of the structural integrity (such as the addition of shared variables). No experience was gained in this area, though.

However, the extreme testability and configurability of the approach was a real pleasure during implementation and debugging. As we implemented our designs from the bottom up, we were able to extensively exercise each module as we went, especially the corner cases which always have to be coded for but which are often very difficult to test in a monolithic system. This gave us greater confidence in the base we were building, and debugging higher level modules was not significantly more difficult than low level modules, even though they had a greater number of dependencies.

References

1. M. J. Bach and S. J. Burroff, Multiprocessor Unix Operating Systems, *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, October 1984, pages 1733-1749.
2. Steve Boettner, personal communication, 1986.
3. DoD 5200.28-STD, *Department of Defense Trusted Computer System Evaluation Criteria* ("Orange Book"), December 1985.
4. Robert. A. Gingell, Joseph. P. Moran, William. A. Shannon, Virtual Memory Architecture in SunOS, *USENIX Proceedings*, Summer 1987, pages 81-94.
5. G. Grossman, How Secure is 'Secure'?, *Unix Review*, August 1986, page 60.
6. Graham Hamilton and Daniel S. Conde, An Experimental, Symmetric Multiprocessor Ultrix Kernel, *USENIX Proceedings*, Winter 1988, pages 283-290.
7. Anita K. Jones, The Object Model: A Conceptual Tool for Structuring Software, *Operating Systems: An Advanced Course*, Springer-Verlag 1979, pages 8-16.

8. P. A. Janson, Using Type-Extension to Organize Virtual Memory Mechanisms, *Operating Systems Review*, Vol. 15, No. 4, October 1981, pages 6-38.

9. Butler W. Lampson, Hints for Computer System Design, *Proceedings of the 9th ACM Symposium on Operating Systems Principles,* October 1983.

10. David L. Parnas, Designing Software for Ease of Extension and Contraction, *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 2, March 1979, pages 128-138.

11. J. Eric Roskos, A Comparison of Two Synchronization Primitives in an Operating System for Parallel Processing Applications, *Proceedings of the 1986 International Conference on Parallel Processing.*

12. W. Olin Sibert, Deborah D. Downs, Holly M. Traxler, Grant M. Wagner, Jeffrey J. Glass, UNIX and B2: Are They Compatible?, Draft (June 1987), National Computer Security Center.

13. Ursula Sinkewicz, A Strategy for SMP Ultrix, *USENIX Proceedings*, Summer 1988, pages 203-212.

14. Harold S. Stone, *High-Performance Computer Architecture*, Addison-Wesley, 1987, pages 336-338.

15. Bjarne Stroustrup, Possible Directions for C++, *Proceedings USENIX C++ Workshop*, 1987.

16. Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, Robert Baron, The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System, *Proceedings of the Eleventh ACM Symposium on Operating System Principles,* November 1987.

Evolving the UNIX System Interface to Support Multithreaded Programs

Paul R. McJones and Garret F. Swart
DEC Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301

Abstract

Allowing multiple threads to execute within the same address space makes it easier to write programs that deal with related asynchronous activities and that execute faster on shared-memory multiprocessors. Supporting multiple threads places new constraints on the design of operating system interfaces. We present several guidelines for designing or redesigning interfaces for multithreaded clients. We show how these guidelines were used to design an interface to UNIX[1]-compatible file and process management facilities in the Topaz operating system. Two implementations of this interface are in everyday use: a native one for the Firefly multiprocessor, and a layered one running within a UNIX process.

1. Introduction

Most existing general-purpose operating systems place in one-to-one correspondence virtual address spaces and threads, where by a thread we refer to the program counter and other state recording the progress of a sequential computation. This one-to-one correspondence between address spaces and threads makes it more difficult to construct applications dealing with asynchrony and to exploit the speed of multiprocessors. To address these problems, several newer operating systems allow multiple threads within a single virtual address space. The existence of multiple threads within an address space places additional constraints on the design of operating system interfaces. In this paper we present several guidelines that we used to design the multithreaded operating system interface of the Topaz system built at DEC's Systems Research Center (SRC). We show how we used these guidelines to evolve the Topaz interface from the 4.2BSD UNIX [12] system interface. We believe the guidelines will be useful for adding multithreading to other operating systems.

One implementation of Topaz runs as the native operating system on SRC's Firefly multiprocessor [19] and allows concurrent execution on separate processors of multiple threads within the same address space. A second implementation of Topaz is layered on 4.2BSD UNIX; it uses multiprogramming techniques to create multiple threads within a single UNIX process. Both implementations make it convenient to compose single-threaded UNIX programs and multithreaded programs using the standard UNIX process composition mechanisms [14].

Topaz is an extension of the architecture of an existing system rather than an entirely new design because of the dual role it plays at SRC. Topaz serves both as the base for research into distributed systems and multiprocessing and also as the support for SRC's current computing needs, which are mainly document preparation, electronic mail, and software development. When experimental software can be put into everyday use on the same system that runs existing tools and applications, it is easier to get relevant feedback on that software.

[1]UNIX is a trademark of AT&T Bell Laboratories

There were several reasons for choosing UNIX in particular as an architectural starting point. The machine-independence of UNIX left the way open for future work at SRC on processor design. UNIX also offered a large set of tools and composition mechanisms, and a framework for exchanging ideas about software throughout the research community.

Section 2 gives a brief overview of Topaz, to set the stage for the rest of the paper. Sections 3 and 4 constitute the heart of the paper: our guidelines for multithreaded interfaces and our use of those guidelines in designing the Topaz operating system interface. Section 5 draws some conclusions about the approach taken in Topaz.

2. Topaz Overview

One way of viewing Topaz is as a hybrid of Berkeley's 4.2BSD UNIX [12] and Xerox's Cedar [17]. Topaz borrows the 4.2BSD file system semantics and large-grain process structure, populates these processes (address spaces) with Cedar-like threads, and interconnects them with Cedar-like remote procedure call [5]. Topaz allows single-threaded programs using the standard 4.2BSD system interface and multithreaded programs using a new Topaz operating system interface to run on the same machine, to share files, to send each other signals, and to run each other as processes.

A Topaz address space has all of the state components that a UNIX process has, such as virtual memory, a set of open files, a user id, and signal-handling information. While a UNIX process has only one stack and set of registers, a Topaz address space has a separate stack and set of registers for each thread of control living in that address space.

A Topaz programmer can use threads for fine-grained cooperation, as is done in the Cedar system. Unlike a Cedar programmer, a Topaz programmer can also use multiple address spaces to separate programs of different degrees of trustworthiness. Many Topaz address spaces contain long-running servers handling remote procedure calls from other address spaces on the same or different machines.

Multiple threads address a problem different from the one addressed by the shared memory segments provided by some versions of UNIX, such as System V [2]. While shared segments are useful in allowing separately developed application programs to have access to a common data structure, as for example a database buffer pool, multiple threads are intended to be a "lightweight" control structure for use within a single program. One example is that the Topaz remote procedure call mechanism executes concurrent incoming calls in separate threads. Another example is that the Topaz window system uses several threads in a pipeline arrangement to spread the work of transporting and processing painting requests over several CPUs.

Modeling threads as separate UNIX processes would mean that threads could not freely share open file descriptors, since UNIX only allows these descriptors to be inherited by a child process from its parent process. It would also be difficult to share pointer-containing data structures among threads modeled as separate UNIX processes, since a pointer into the stack segment would have a different meaning in each process. Many Topaz applications create dozens or hundreds of threads. This would be slow and extravagant of kernel resources if each was a full UNIX process, even if most of the virtual memory could be shared.

A Topaz application is written as if there is a processor for every thread; the implementation of Topaz assigns threads to actual processors. Threads sharing variables must therefore explicitly synchronize. The synchronization primitives provided (mutexes, conditions, and semaphores) are derived from Hoare's

monitors [6], following the modifications of Mesa [9]; the details are described by Birrell et al. [4].

Support for multiprocessors in UNIX has evolved over a number of years. Early multiprocessor implementations of UNIX allowed concurrent execution of single-threaded processes but didn't support multiple threads. Many of these implementations serialized execution within the system kernel; Bach and Buroff [3] describe one of the first implementations to allow concurrency within the kernel. Several current systems, such as Apollo's Concurrent Programming Support [1] and Sun's "lightweight process" (lwp) facility [7], support multiple threads within a UNIX process, but can't assign more than one thread within an address space to a processor at any one time. Like the Firefly implementation of Topaz, C-MU's Mach [13] supports concurrent execution of threads within an address space on a multiprocessor. The approach taken by Apollo, Sun, and Mach in adding threads to UNIX is to minimize the impact on the rest of the system interface, to make it easier to add the use of multiple threads to large existing programs. In contrast, the approach taken in Topaz is to integrate the use of threads with all the other programming facilities.

3. Guidelines for Multithreaded Interfaces

By a *multithreaded interface* we mean one usable by multithreaded clients. Good interface design is a challenging art, and has a whole literature of its own (for example, see Parnas [11] and Lampson [8]). In this section we present three guidelines abstracted from our experience designing the Topaz operating system interface.

Our first guideline addresses an aspect of interface design that is complicated by multiple threads: avoiding unnecessary serialization to mutable state defined by the interface. Our second guideline addresses an aspect of interface design that is simplified by multiple threads: dealing with asynchrony without resorting to ad hoc techniques. Our third guideline addresses the problem of cancelling undesired computations in a multithreaded program.

3.1. Sharing Mutable State

It is not uncommon for a single-threaded interface to reference a state variable that affects one or more procedures of the interface. The purpose is often to shorten calling sequences by allowing the programmer to omit an explicit argument from each of a sequence of procedure calls in exchange for occasionally having to set the state variable. To avoid interference over such a state variable, multiple client threads must often serialize their calls on procedures of the interface even when there is no requirement for causal ordering between the threads.

One example of interference caused by an interface state variable is the stream position pointer within a UNIX open file [14]. The pointer is implicitly read and updated by the stream-like **`read`** and **`write`** procedures and is explicitly set by the **`seek`** procedure. If two threads use this interface to make independent random accesses to the same open file, they have to serialize all their **`seek-read`** and **`seek-write`** sequences. Another example is the UNIX library routine **`ctime`**, which returns a pointer to a statically allocated buffer containing its result and so is not usable by concurrent threads.

While it is important to avoid unnecessary serialization of clients of an interface, serialization within the implementation of a multithreaded interface containing shared data structures is often necessary. This is to be expected and will often consist of fine-grain locking that minimizes interference between threads.

We can think of four basic approaches to designing multithreaded interfaces so as to minimize the

possibility of interference between client threads over shared mutable state:

1. Make it an argument. This is the most general solution, and has the advantage that one can maintain more than one object of the same type as the shared mutable state being replaced. In the file system example, passing the stream position pointer as an argument to `read` and `write` solves the problem. Or consider a pseudo-random number generator with a large amount of hidden state. Instead of making the client synchronize its calls on the generator, or even doing the synchronization within the generator, either of which may slow down the application, a better solution is to store the generator state in a record and to pass a pointer to this record on each call of the generator.

2. Make it a constant. It may be that some state component need not change once an application is initialized. An example of this might be the user on whose behalf the application is running.

3. Let the client synchronize. This is appropriate for mutable state components that are considered inherently to affect an entire application, rather than to affect a particular action being done by a single thread.

4. Make it thread-dependent, by having the procedure use the identity of the calling thread as a key to look up the variable in a table. Adding extra state associated with every thread adds to the cost of threads, and so should not be considered lightly. Having separate copies of a state variable can also make it more difficult for threads to cooperate in manipulating a single object.

It is a matter of judgment which of these techniques to use in a particular case. We used each of the four in designing the Topaz operating system interface. Sometimes providing a combination offers worthwhile flexibility. For example, a procedure may take an optional parameter that defaults to a value set at initialization time. Also, it is possible for a client to simulate thread-dependent behavior by using a procedure taking an explicit parameter in conjunction with an implementation of a per-thread property list (set of tag-value pairs).

3.2. Avoiding Ad Hoc Multiplexing

Although most operating systems provide only a single thread of control within each address space, application programs must often deal with a variety of asynchronous events. As a consequence, many operating systems have evolved a set of ad hoc techniques for multiplexing the single thread within an address space. These techniques have the disadvantage that they add complexity to applications and confuse programmers. To eliminate the ad hoc techniques, multiple threads can be used, resulting in simpler, more reliable applications.

The aim of all the ad hoc multiplexing techniques is to avoid blocking during a particular call on an operating system procedure when the client thread could be doing other useful work (computing, or calling a different procedure). Most of the techniques involve replacing a single operating system procedure that performs a lengthy operation with separate methods for initiating the operation and for determining its outcome. The typical methods for determining the outcome of such an asynchronous operation include:

Polling. Testing whether or not the operation has completed, as by checking a status field in a control block that is set by the operation. Polling is useful when the client thread wants to overlap computation with one or more asynchronous operations. The client must punctuate its computation with periodic calls to the polling procedure; busy waiting results when the client has no other useful computation. Note that busy waiting is undesirable only when there is a potential for the processor to be used by another process.

Waiting. Calling a procedure that blocks until the completion of a specified operation, or more usefully one of a set of operations. Waiting procedures are useful when the client

thread is trying to overlap a bounded amount of computation with one or more asynchronous operations, and must avoid busy waiting. The use of a multiway waiting procedure hinders program modularity, since it requires centralized knowledge of all asynchronous operations initiated anywhere in the program.

Interrupts. Registering a procedure that is called by borrowing the program counter of the client thread, like a hardware interrupt. Interrupts are useful in overlapping computation with asynchronous operations. They eliminate busy waiting and the inconsistent response times typical of polling. On the other hand, they make it difficult to maintain the invariants associated with variables that must be shared between the main computation and the interrupt handler.

The techniques are often combined. For example, 4.2BSD UNIX provides polling, waiting, and interrupt mechanisms. When an open file has been placed in non-blocking mode, the **`read`** and **`write`** operations return an error code if a transfer is not currently possible. Non-blocking mode is augmented with two ways to determine a propitious time to attempt another transfer. The **`select`** operation waits until a transfer is possible on one of a set of open files. When an open file has been placed in asynchronous mode, the system sends a signal (software interrupt) when a transfer on that file is possible.

When multiple threads are available, it is best to avoid all these techniques and to model each operation as a single, synchronous procedure. This is simple for naive clients, and allows more sophisticated clients to use separate threads to overlap lengthy system calls and computation.

3.3. Cancelling Operations

Many application programs allow the cancellation of a command in progress. For example, the user may decide not to wait for the completion of a computation or the availability of a resource. In order to allow prompt cancellation, an application needs a way of notifying all the relevant threads of the change in plans.

If the entire application is designed as one large module, then state variables, monitors, and condition variables may be enough to implement cancellation requests. However, if the application is composed of lower-level modules defined by interfaces, it is much more convenient to be able to notify a thread of a cancellation request without regard for what code the thread is currently executing.

The Topaz system provides the *alert* mechanism [4] for this purpose; it is similar to Mesa's *abort* mechanism [9]. Sending an alert to a thread simply puts it in the alerted state. A thread can atomically test-and-clear its alerted status by calling a procedure **`TestAlert`**. Of course this is a form of polling, and isn't always appropriate or efficient. To avoid the need to poll, there exist variants of the procedures for waiting on condition variables and semaphores. These variants, **`AlertWait`** and **`AlertP`**, return prematurely with a special indication if the calling thread is already in, or enters, the alerted state. The variants also clear the alerted status. We refer to the procedures **`TestAlert`**, **`AlertWait`**, and **`AlertP`**, and to procedures that call them, as *alertable*.

What then is the effect of alerts on interface design? Deciding which procedures in an interface should be alertable requires making a trade-off between the ease of writing responsive programs and the ease of writing correct programs. Each call of an alertable procedure provides another point at which a computation can be cancelled, and therefore each such call also requires the caller to design code to handle the two possible outcomes: normal completion and an alert being reported. We have formulated the following guidelines for using alerts in an effort to define the minimum set of alertable procedures necessary to allow top-level programs to cancel operations:

1. Only the owner of a thread, that is the program that forked it, should alert the thread. This is

because an alert carries no parameters or information about its sender. A corollary is that a procedure that clears the alerted status of a thread must report that fact to its caller, so that the information can propagate back to the owner.

2. Suppose there is an interface M providing a procedure P that does an unbounded wait, that is a wait whose duration cannot be bounded by appeal to M's specification alone. Then M should provide alertable and nonalertable variants of the procedure, just as Topaz does for waits on condition variables and semaphores. (The interface might provide either separate procedures or one procedure accepting an "alertable" Boolean parameter.) A client procedure Q should use the alertable variant of P when it needs to be alertable itself and cannot determine a bound on P's wait.

3. A procedure that performs a lengthy computation should follow one of two strategies. It can allow partial operations, so that its client can decompose a long operation into a series of shorter ones separated by an alert test. Or it can accept an "alertable" Boolean parameter that governs whether the procedure periodically tests for alerts.

If all interfaces follow these rules, a main program can always alert its worker threads with the assurance that they will eventually report back. The implementation of an interface might choose to call alertable procedures in more cases than required by the second guideline, gaining quicker response to alerts at the cost of more effort to maintain its invariants.

4. Topaz Operating System Interface

Topaz programs are written in Modula-2+ [16], which extends Wirth's Modula-2 [20] with concurrency, exception handling, and garbage collection. The facilities of Topaz are provided through a set of interfaces, each represented by a Modula-2+ definition module.

This section describes the Topaz OS interface, which contains the file system and process (address space) facilities. We focus here on how the presence of multiple threads affected the evolution of the OS interface from the comparable 4.2BSD UNIX facilities. More information about the Topaz OS interface can be found in its reference manual [10].

4.1. Reporting Errors

A UNIX system call reports an error by storing an error number in the variable `errno` and then returning the value -1. The variable `errno` causes a problem for a multithreaded client, since different values could be assigned due to concurrent system calls reporting errors. (Another source of confusion results from system calls that can return -1, e.g., `nice` or `ptrace`.)

A workable solution would be for every system call that could report an error to return an error code via a result parameter. We chose to use Modula-2+ exceptions instead, for reasons that had little to do with the presence of multiple threads. It is worth noting that exceptions have the advantage over return codes that they can't be accidentally ignored, since an exception which has no handler results in abnormal termination of the program. This problem is serious enough that UNIX uses signals to report certain synchronous events; for example `SIGPIPE` is raised when a process writes to a pipe whose reading end is no longer in use.

A Modula-2+ procedure declaration may include a `RAISES` clause enumerating the exceptions the procedure may raise. The declaration of an exception may include a parameter, allowing a value to be passed to the exception handler. Most procedures in the Topaz operating system interface can raise the exception `Error`, which is declared with a parameter serving as an error code, analogous to the UNIX

error number. Topaz defines the exception `Alerted` for reporting thread alerts (discussed in Section 3.3). Each procedure in the Topaz operating system interface that may do an unbounded wait includes `Alerted` in its `RAISES` clause. As described in Section 4.3, Topaz also uses exceptions to report synchronous events such as hardware traps.

4.2. File System

A UNIX process contains several components of mutable file system state that would cause problems for multithreaded programs, including the working directory, the table of file descriptor references, and the stream position pointer inside each file descriptor. The Topaz design has made adjustments for each of these.

A UNIX path name is looked up relative to the file system root if it begins with "/"; otherwise it is looked up relative to the working directory. Each process has its own working directory, which is initially equal to the parent's and may be changed using the `chdir` system call. Since looking up a short relative path name can be significantly faster than looking up the corresponding full path name, some UNIX programs use the working directory as a sort of "cursor", for example when enumerating a subtree of the file system. To facilitate multithreaded versions of such programs (and modular programming in general), Topaz parameterizes the notion of working directory. The `OpenDir` procedure accepts the path name of a directory, and returns a handle for that directory. Every procedure that accepts a path name argument also accepts a directory handle argument that is used when the path name doesn't begin with "/". The distinguished directory handle `NIL` can be used to refer to the initial working directory supplied when the process was created.

Part of the state maintained by UNIX for each process is a table with an entry for each open file held by the process. An application program uses small nonnegative integer indices in this table to refer to open files. In a multithreaded application it is desirable to avoid the need to serialize sequences of operations affecting the allocation of table entries (e.g., `open`, `dup`, and `close`). To achieve this goal, the table indices should be treated as opaque quantities: it should not be assumed that there is a deterministic relationship between successive values returned by operations such as `open`. (Single-threaded UNIX programs actually depend on being able to control the allocation of table indices when preparing to start another program image. Topaz avoids this dependency, as described in Section 4.4.)

Recall from the example in Section 3.1 that the stream position pointer in a UNIX file descriptor causes interference when threads share the descriptor. Topaz still implements these pointers so that Topaz and UNIX programs can share open files, but to allow multiple threads to share a file descriptor without having to serialize, Topaz provides additional procedures `FRead` and `FWrite` that accept a file position as an extra argument.

The 4.2BSD UNIX file system interface contains a number of ad hoc multiplexing mechanisms that are described in Section 3.2. These mechanisms allow a single-threaded UNIX process to overlap computation and input/output transfers that involve devices such as terminals and network connections. Topaz simply eliminates these mechanisms (non-blocking mode, the `select` procedure, and asynchronous mode) and substitutes `Read` and `Write` procedures that block until the transfer is complete. `Read` and `Write` are alertable when a transfer is not yet possible. Note that Topaz violates guideline 2 of Section 3.3 by not providing nonalertable variants of `Read` and `Write`. For completeness, Topaz provides a `Wait` procedure that waits until a specified open file is ready for a transfer.

4.3. Signals

A UNIX signal is used to communicate an event to a process or to exercise supervisory control over a process, such as termination or temporary suspension. A UNIX signal communicates either a synchronous event (a trap, stemming directly from an action of the receiving process) or an asynchronous one (an interrupt, stemming from another process, user, or device).

UNIX models signal delivery on hardware interrupts. A process registers a handler procedure for each signal it wants to handle. When a signal is received, the current computation is interrupted by the creation of an activation record for the handler procedure on the top of the stack of the process. This handler procedure may either return normally, resulting in the interrupted computation continuing, or may do a "long jump", unwinding the stack to a point specified earlier in the computation. If a signal is received for which no handler procedure was registered, a default action takes place. Depending on the signal, the default action is either to do nothing, to terminate the process, to stop the process temporarily, or to continue the stopped process. Following the hardware interrupt model, 4.2BSD UNIX allows each signal to be ignored or temporarily masked.

Topaz signals are patterned after UNIX signals, and in fact Topaz and UNIX programs running on the same machine can send each other signals. However, UNIX signal delivery is another ad hoc way of multiplexing the single program counter of a process. Trying to use interrupt-style signal delivery in a multithreaded environment leads to problems. Which thread should receive the signal? What does a signal handler procedure do if it needs to acquire a lock held by the thread it has interrupted? Rather than answering these questions, we avoided them.

A Topaz process can specify that it wants to handle a particular signal, but it doesn't register a handler procedure. Instead, it arranges for one of its threads to call `WaitForSignal`. This procedure blocks until a signal arrives, then returns its signal number. The calling thread then takes whatever action is appropriate, for example initiating graceful shutdown. `WaitForSignal` takes a parameter that specifies a subset of the handled signals, so a program may have more than one signal-handling thread. The set of signals that it makes sense to handle is smaller in Topaz than in UNIX, since those used as part of various UNIX ad hoc multiplexing schemes (e.g., `SIGALRM`, `SIGURG`, `SIGIO`, and `SIGCHLD`) are never sent to multithreaded processes. Topaz provides the same default actions as UNIX for signals not handled by the process. The decision about which signals to handle and which to default is necessarily global to the entire process; any dynamic changes must be synchronized by the client.

UNIX system calls that do unbounded waits (e.g., reading from a terminal or waiting for a child process to terminate) are interruptible by signals. But this interruptibility leads to difficulties that are avoidable in the multithreaded case. A client program will normally want to restart a system call interrupted by a signal that indicates completion of some asynchronous operation, but will probably not want to restart a system call interrupted by a signal that indicates a request for cancellation of a computation. Different versions of UNIX have tried different approaches to the restartability of system calls. In Topaz, there is no need for signal delivery itself to interrupt any system call. The signal handling thread may decide to alert one or more other threads, which raises an `Alerted` exception in a thread doing an unbounded wait in a system call.

Instead of using signals to report synchronous events, Topaz uses Modula-2+ exceptions. For example, the `AddressFault` exception is raised when a thread dereferences an invalid address. Since the contexts statically and dynamically surrounding where an exception is raised determine what handler is invoked for that exception, different threads can have different responses.

Modula-2+ exceptions are based on a termination model: scopes between the points where an exception is raised and where it is handled are *finalized* (given a chance to clean up and then removed from the stack) before the handler is given control. While this model has proven its worth in constructing large modular systems, it does lead to a complication involving traps. An exception is an appropriate way to report a trap considered to be an error, but isn't appropriate for a trap such as breakpoint or page fault whose handler wishes to resume. Topaz therefore provides a lower-level trap mechanism that suspends a thread at the point of the trap and then wakes up a trap-handling thread. The trap-handling thread either converts the trap to an exception in the trapping thread, or handles it and restarts the thread, as appropriate.

4.4. Process Creation

UNIX provides simple but powerful facilities for creating processes and executing program images, involving three main system calls: `fork`, `wait`, and `exec`. `fork` creates a child process whose memory, set of open files, and other system-maintained state components are all copied from the calling process. `wait` waits for the next termination of a child of the calling process; there is also a nonblocking form of `wait` and a signal `SIGCHLD` sent by the system whenever a child process terminates. `exec` overlays the address space of the calling process with a new program image.

Typically UNIX programs use these facilities in one of two stylized ways. One is to create an extra thread of control; for example, a "terminal emulator" program uses a pair of processes for full-duplex communication with a remote system. The other is to run a new program image; for example, the shell runs each command in a new child process.

In Topaz, the way to create an extra thread of control is simply to fork a new thread within the same process. Topaz must still provide a mechanism for running a new program image, and the UNIX method won't do. In UNIX, the parent calls `fork`; the child (initially executing the same program image as the parent) makes any necessary changes to its process state (e.g., opening and closing files or changing the user id), and finally calls `exec` to overlay itself with the new program image. Since the `fork-exec` sequence involves a large amount of shared mutable state (the entire child process), it isn't surprising that it doesn't work for Topaz. Would `fork` copy all threads? In the Apollo and Mach systems, only the thread that calls `fork` is copied [18]. But what happens if locks were held by other threads in the parent process? If `fork` copied all threads, what would it mean to copy a thread blocked in a system call?

To address this problem, Topaz replaces `fork` and `exec` with a single new procedure `StartProcess`, which accepts as parameters all the modifiable components of a process state (namely anything that could be changed between a call to `fork` and the subsequent call to `exec`, such as the set of open files and the user id). Topaz also replaces `wait` with `WaitForChild`, which waits for the termination of a specific child process.

There are several reasons for not merging the functions of `StartProcess` and `WaitForChild` into a single procedure that would block until termination of the child process. As it stands, `StartProcess` returns the process identifier of the new process and `WaitForChild` returns a status value indicating whether the child terminated or temporarily stopped. These features allow a process to be observed and controlled while it executes, in a way compatible with 4.2BSD UNIX job control.

4.5. Other Process State

The preceding subsections describe how Topaz treats many of the mutable state components of a UNIX process. For completeness, here is how the remaining components are treated.

Process Group.	Constant (fixed at the time the process is created).
User identity.	Constant/parameterized. Most processes run with a fixed user identity. A super-user program, usually a server, acting on behalf of many users may pass an additional parameter on each call giving the user identity on whose behalf it is acting.
Control Terminal.	Constant.
UMask.	Client-synchronized. (The umask is used to set the access control bits when a file is created.)
Priority.	Thread-dependent.

4.6. Summary of control-structure changes

We can summarize the changes to the control structure of the Topaz operating system interface as follows:

- A computation is overlapped by performing it in a separate thread.
- An asynchronous event is delivered by unblocking a client thread.
- A synchronous event is delivered by returning a value or raising an exception in the responsible thread.

5. Conclusions

The implementation of Topaz for SRC's Firefly multiprocessor has been in daily use since the spring of 1986, and the version of the multithreaded operating system interface described here has been in use since the spring of 1987. A number of multithreaded application programs and servers have been written for Topaz.

We consider both the idea of a multithreaded extension of UNIX and our new operating system interface to be successes. Users have access to the large collection of UNIX application software, are free to investigate the consequences of multiple threads and remote procedure call for building new applications, and are often able to use UNIX and Topaz applications together. For example most Firefly users use the standard C shell both interactively and through shell scripts to run a mixture of UNIX and Topaz applications. Most Fireflies run a Topaz "distant process" server that allows a user to run arbitrary processes on idle machines throughout the local network, for example to run a parallel version of the UNIX make [15].

Supporting multiple threads instead of a single thread adds little to the inherent execution-time cost of the system-call interface. And of course good speedups are possible when clients take advantage of opportunities for concurrent execution. For example, a Topaz command called *updatefs* compares two file systems, bringing one up-to-date with respect to the other. Changing *updatefs* to use concurrent threads to traverse the two trees and compare file modification times resulted in a speed-up of 3 to 4 (on a 5-processor Firefly).

The implementation of Topaz layered on UNIX allows us to write servers (e.g., for remote file access and remote login) that run on Fireflies and on VAX/UNIX systems with few or no source changes. It also provides a way for us to export Topaz application software to UNIX sites.

We believe that the guidelines presented in this paper will be useful in designing other interfaces for use by multithreaded programs. The extra parameterization necessary to avoid shared mutable state might not always be useful in a purely sequential program, although it is likely to ease the construction of modular programs. Eliminating ad hoc multiplexing has the property that the resultant interface can be viewed as appropriate for use by a purely sequential program, so in some sense no complexity is added for use by a multithreaded program. Making operations cancellable is often an externally imposed requirement; the multithreaded approach avoids many of the problems with interrupts and restartability.

Acknowledgments

The Topaz system as a whole is the work of many people at SRC. The guidelines for alertability in Section 3.3 resulted from discussions with Andrew Birrell and Roy Levin. The OS interface and its reference manual profited from Andy Hisgen's careful reading and comments. Andrew Birrell and Michael Schroeder provided valuable advice on the overall design. John DeTreville, Cynthia Hibbard, Michael Schroeder, and Roger Needham provided comments that improved this paper.

References

1. Apollo Computer Inc. *Concurrent Programming Support (CPS) Reference.* 330 Billerica Road, Chelmsford, MA 01824, 1987.

2. AT&T. *System V Interface Definition, Issue 2.* Customer Information Center, P.O. Box 19901, Indianapolis, IN 46219, 1986.

3. Bach, M. J. and Buroff, S. J. "Multiprocessor UNIX operating systems". *AT&T Bell Laboratories Technical Journal 63*, 8 (Oct. 1984), 1733-1749.

4. Birrell, A. D., Guttag, J. V., Horning, J. J., and Levin, R. Synchronization primitives for a multiprocessor: a formal specification. Proceedings of the Eleventh Symposium on Operating System Principles, ACM, New York, Nov., 1987, pp. 94-102.

5. Birrell, Andrew D. and Nelson, Bruce Jay. "Implementing remote procedure calls". *ACM Transactions on Computer Systems 2*, 1 (Feb. 1984), 39-59.

6. Hoare, C. A. R. "Monitors: an operating system structuring concept". *Comm. ACM 17*, 10 (Oct. 1974), 549-557.

7. Jonathan Kepecs. Lightweight processes for UNIX implementation and applications. USENIX Association Conference Proceedings, June, 1985, pp. 299-308.

8. Lampson, Butler W. "Hints for computer system design". *IEEE Software 1*, 1 (Jan. 1984), 11-28.

9. Lampson, Butler W. and Redell, David D. "Experience with processes and monitors in Mesa". *Comm. ACM 23*, 2 (Feb. 1980), 105-117.

10. McJones, Paul R. and Swart, Garret F. The Topaz operating system programmer's manual. In *Evolving the UNIX System Interface to Support Multithreaded Programs, Research Report #21*, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, 1987.

11. Parnas, D. L. "On the criteria to be used in decomposing systems into modules". *Comm. ACM 15*, 12 (Dec. 1972), 1053-1058.

12. Quarterman, John S., Silberschatz, Abraham, and Peterson, James L. "4.2BSD and 4.3BSD as examples of the UNIX system". *Comput. Surv. 17*, 4 (Dec. 1985), 379-418.

13. Rashid, Richard F. "Threads of a new system". *UNIX REVIEW 4*, 8 (Aug. 1986), 37.

14. Ritchie, Dennis M. and Thompson, Ken. "The UNIX time-sharing system". *Comm. ACM 17*, 7 (July 1974), 365-375.

15. Roberts, Eric S. and Ellis, John R. parmake and dp: Experience with a distributed, parallel implementation of make. Proceedings of the Second IEEE-CS Workshop on Large Grained Parallelism, Oct., 1987.

16. Rovner, Paul. "Extending Modula-2 to build large, integrated systems". *IEEE Software 3*, 6 (November 1986), 46-57.

17. Swinehart, Daniel C., Zellweger, Polle T., and Hagmann, Robert B. The structure of Cedar. Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments, ACM, New York, June, 1985, pp. 230-244.

18. Tevanian, Avadis, Jr., Rashid, Richard F., Golub, David B., Black, David L., Cooper, Eric, and Young, Michael W. Mach Threads and the Unix Kernel: The battle for control. USENIX Association Conference Proceedings, Phoenix, June, 1987, pp. 185-197.

19. Thacker, Charles P. and Stewart, Lawrence C. Firefly: a multiprocessor workstation. Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems, ACM and IEEE Computer Society, Oct., 1987.

20. Wirth, Niklaus. *Programming in Modula-2*. Springer-Verlag, 1985.

Variable Weight Processes with Flexible Shared Resources *

Ziya Aral † James Bloom ‡ Thomas Doeppner ‡ Ilya Gertner †
Alan Langerman † Greg Schaffer †

Encore Computer Corporation and Brown University

Abstract

Traditional UNIX processes are inadequate for representing multiple threads of control in parallel programs. They are inflexible in their resource allocation, unable to cleanly share system resources, and they carry a heavy overhead. Some new operating systems, such as MACH, split the process into multiple light-weight threads of execution and a task which defines their resource set. This paper addresses the same problem within the confines of UNIX process semantics. It partitions the existing concept of a process into a new "variable weight" process and several independent "system resource descriptors". Processes pay creation and maintenance costs only for resources they wish to keep private. Surprisingly few changes to the kernel are needed to achieve this effect. The design can properly be considered as a simple concurrency extension to UNIX. Variable-weight processes also enhance the effectiveness of the cooperating process paradigm and so are broadly applicable to both uniprocessor and multiprocessor UNIX implementations.

A prototype kernel has been implemented on Multimax^{TM}, a shared-memory multiprocessor. Preliminary experience indicates that relatively few changes to the process structures in UNIX make this strategy incrementally applicable to a range of UNIX variants.

Introduction

The design of a symmetric multi-threaded UNIX kernel on a shared memory multiprocessor is by now a problem that is well understood [2,1,13,11,8]. Much less clear are the appropriate abstractions for scheduling and resource management that should be provided by the kernel to support both sequential and multi-threaded programs. What are the characteristics of a system that gives the full capabilities of the multiprocessor hardware to the user, yet remains a system that most would agree is UNIX? In fact, why do we want a system that is UNIX, at all?

UNIX semantics are very well understood by a large number of people. There is a lot of documentation and instructional material on UNIX. There are a large number of UNIX implementations; UNIX has become something of a commercial standard for large systems. Finally, UNIX has proven to be remarkably adaptable to the demands of over a decade of advances in computer science. If support for parallelism can be integrated into UNIX in a clean way so as not to interfere with existing

*This research was supported in part by the Office of Naval Research under Contract No. N00014-88-K-0406; and in part by the Defense Advanced Research Projects Agency (DoD) through ARPA Order No. 5875, monitored by Space and Naval Warfare Systems Command under Contract No. N00039-86-C-0158; and through ARPA Order No. 6320, monitored by the Office of Naval Research under Contract No. N00014-83-K-0146.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

†Encore Computer Corporation, 257 Cedar Hill Street, Marlborough, Ma. 01752-3004.

‡Department of Computer Science, Brown University, Providence, RI 02912-1910.

features and yet be in the style of those existing features, the task of adopting concurrency features is made far easier. In this paper we argue that UNIX may be easily modified to support concurrent programming, both on uniprocessors and on shared-memory multiprocessors.

In both its AT&T and Berkeley incarnations, UNIX's most obvious shortcoming with respect to multiprocessing is the cost of processes. Processes are the most fundamental abstraction of UNIX, and are the only OS-supported execution units. They are, however, very expensive to create and to schedule. Thus it is costly for a programmer to make use of multiple processors, since doing so requires the use of multiple processes.

This paper assumes that concurrency support must address processes and thus must include kernel modifications. We have much experience in supporting multiple threads of control at the user level (using user-level schedulers) and do not question the usefulness of this approach [5,10]. There will always be a need to support application-specific approaches to concurrency that go beyond what the OS can be expected to provide. The exact division of labor between user-level and kernel-level concurrency support is an important topic in its own right, which we will address in the future. However, the fact that these packages must overcome deficiencies in UNIX partly motivates this work.

Most versions of UNIX support some form of memory sharing among processes, a feature necessary for multiprocessor applications. Memory is not the only resource that must be made sharable. There is much discussion in the literature of the need for sharing other resources, such as file descriptors and signal handlers [4,9,11,12].

Our work on monitoring and debugging parallel programs leads us to agree [3,7]. Parallel programming involves the cooperation of a number of threads of control. In some cases, these threads need to share all resources, but in other cases there should be restrictions on this sharing. A debugger needs write access to the debuggee's text segment, but the debuggee should have only execute access. Timers may be associated with a single process or a group of processes. Notification of external events such as I/O completions or the typing of interrupt characters in some instances might be sent to each of many potential interested processes, in others be handled by one process on behalf of a group, and in yet others result in the creation of an ad hoc process. The main point is that the resource sharing provided by most, if not all existing UNIX and variant systems is much too inflexible.

Where support for multiple threads of control which share resources exists, it is assumed that these threads are identical. While the notion of *homogeneous* threads of control, each sharing the same system resource set and executing the same code may be adequately supported, the idea of *heterogeneous* threads of control is not. At best, such heterogeneity is achieved only by falling back on multiple processes, with added overhead, complexity, and semantic incompatibilities resulting from it.

In our experience, the typical case is that of multiple threads of control which want to share memory but not necessarily share other resources. Even when working with a homogeneous paradigm, such examples proliferate. A certain degree of support for heterogeneity is almost always useful and often crucial.

UNIX can accommodate the necessary support for parallel processing without drastic changes to the fundamental notion of process. Rather than completely divorce the thread of control from the set of program resources, the approach we describe will permit *variable-weight* processes spanning the spectrum from *homogeneous* threads of control which share all resources, to *heterogeneous* threads of control which share some, but not all resources, to *heavyweight* processes, which share no resources.

This paper focuses on issues of flexible resource sharing for parallel programs. The implementation of a symmetric multi-threaded kernel and the synchronization primitives associated with it is the starting point for our work. Our base operating system is UMAX4.2, a commercially available, multi-threaded variant of 4.2BSD.

To further narrow the domain, we start with the following assumptions:

1. The "cost" of traditional UNIX processes prohibits their use as the basic building block for all but the coarsest parallel programs. This cost is paid not only on start-up, but throughout a process's lifetime in increased kernel memory requirements, increased paging traffic, and longer process-switching times.

2. The notion that every process maps to a separate program with a distinct address space and resource set is not desirable for parallel programs.

3. Existing kernels, whether intended for uniprocessors or multiprocessors, have inadequate and inflexible mechanisms for sharing system resources and their associated overhead among processes.

4. The key to useful concurrency support is the modification of UNIX processes to reduce their cost and promote flexibly shared resources.

5. Where possible, the preservation of UNIX semantics is very important. Further, the preservation of compatible semantics and a common interface for both parallel and serial programs is equally important.

UNIX Processes

UNIX processes represent the wrong model for parallel programming [12]. They are an artifact of a timesharing environment in which processes were expected to map to programs and the priority for the operating system was the segregation of such programs from each other. Concurrency support was explicitly confined to the kernel. The interprocess communication facilities of UNIX were designed to support low-bandwidth channels among independent programs rather than efficient communication within them. Virtual memory was later implemented with the assumption of a separate address space mapped to each process. UNIX's ancillary subsystems, however, have been amenable to modification or replacement over time [14]. The concept of "process" has proven to be more fundamental; it has remained essentially unchanged.

The problems inherent in the process model for multiprocessing are twofold:

1. Traditional processes are inherently "heavyweight", requiring a complete and independent system context including an address space. As a result, they take up significant space, are time-consuming to create and to delete, and take considerable time to switch.

2. Processes are not designed to share resources. [4].

Most parallel variants of UNIX implement facilities to support resource sharing and parallel programming. In their most advanced form, such implementations create a second execution abstraction to coexist with processes and to support shared resources, lower system cost, or both. One proposal introduced *share groups*, whereby processes share a common resource set [4]. While this implementation addresses the issues of a shared address space and I/O descriptors, it takes a narrow view of what system resources should be sharable and does not address at all the "weight" of processes. In fact, it actually adds to that weight by creating the higher-level abstraction of the share group. Furthermore, arbitrary sharing is not supported: share groups do not provide for heterogeneity.

A much more radical approach is taken by MACH. MACH defines a new "lightweight" unit of control, implemented at kernel-level and explicitly designed to support parallel programs [11]. MACH splits the UNIX process into a "thread," which defines a processor context (registers and stack), and a "task," which defines a system context as a collection of system resources. Lightweight control units are implemented by instantiating multiple threads per task. A much broader view of what resources may be shared is taken. All of the threads associated with a task share all of the resources allocated to that task. Conversely, the combination of one thread plus one task is the equivalent of a process and allows emulation of traditional process semantics.

While lightweight threads thus coexist with heavyweight "processes" (literally, single-threaded tasks), some penalty is paid. Such a scheme creates a second set of semantics for threads, incompatible with process semantics, and requires that all threads automatically share all system resources defined by their associated task. In MACH, heterogeneity is supported only by recourse to multiple tasks, the equivalent of multiple "heavyweight" UNIX processes. In addition, MACH is predicated

on an entirely new kernel: to the extent that concurrency is supported, it is not UNIX; to the extent that it is UNIX, no concurrency extensions are available.

We attempt to solve the problems cited above while preserving UNIX process semantics. The motivation for this research was our experience with parallel programming environments, applications, and user-level threads systems. This experience suggested the need not only for greater flexibility in the sharing of resources but also for simplicity and consistency between interfaces for serial and parallel programming. In addition, we saw the need for a general mechanism which could be incrementally developed as further experience with multiprocessing is gained. Surprisingly few changes to UNIX are needed to achieve these goals and the design that is offered can properly be considered as a simple extension to UNIX.

We partition the existing concept of a UNIX process into a new "variable-weight" process and several independent "system resources". We define new primitives for manipulating those resources in order to create a new thread of execution and to selectively manage system resources associated with it. Resources can be arbitrarily shared, or not, by each of these new processes. All existing system interfaces are retained. These processes are as lightweight as MACH threads when they share all resources, and as heavyweight as traditional processes when all resources are private to the process.

This mechanism supports standard UNIX processes and lightweight threads; in addition, it has the flexibility to create heterogeneous threads of execution that share some resources and leave others private.

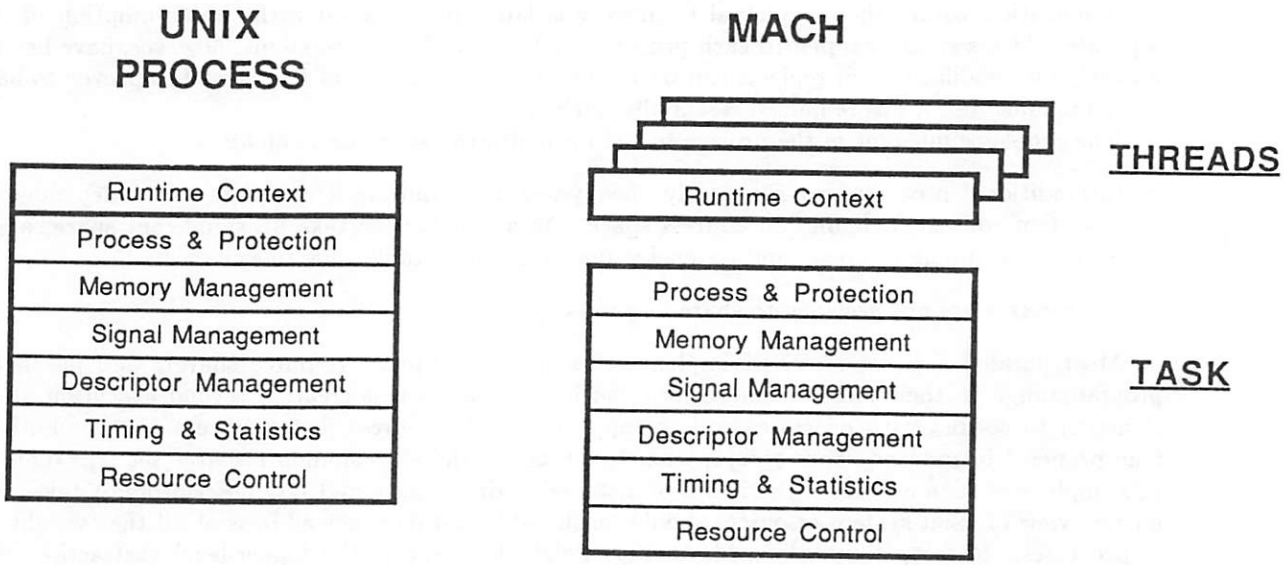


Figure 1: UNIX PROCESS and MACH reconfiguration

nUNIX — A Resource-Oriented Kernel

A prototype kernel, which we will call nUNIX to distinguish it from more conventional variants, implements this capability. The approach may be thought of as being conceptually similar to that of MACH, although the implementation is practically the opposite. Like MACH, this kernel breaks up the process data structures. Instead of breaking out the runtime context from the process descriptor, however, it breaks out the individual resource descriptors. These in turn acquire an existence independent of any particular process.

A UNIX process is described by two data structures, PROC and USER. The partitioning of process information between the two structures is of importance only to the extent that the former contains data which must always remain resident in core while the latter is swappable. For all

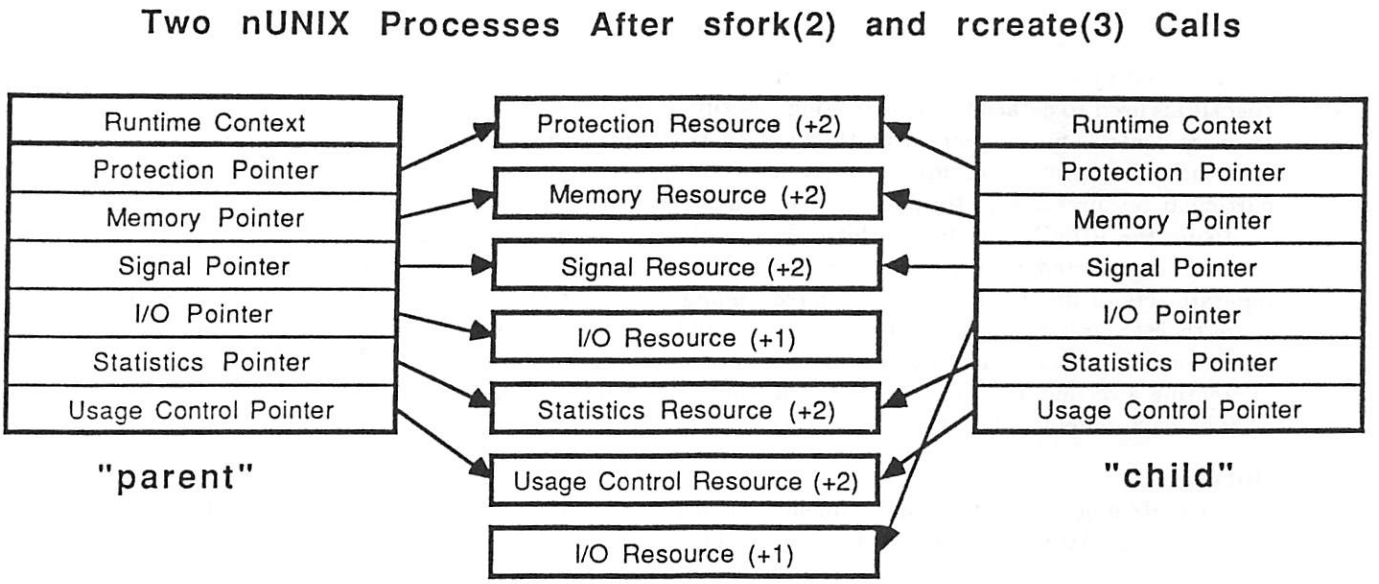


Figure 2: Resource Descriptor PROCESSES

practical purposes, we can abstract a merged structure that defines processes. This structure, in turn, is an amalgam of various independent resources whose only commonality is that together they represent the system environment of the process.

The basis of the nUNIX implementation is the incorporation of an additional level of indirection in the resource descriptor fields of the process data structures. Instead of declaring process resources, such as file descriptors, in-line, the process data structures in nUNIX contain pointers to resource descriptors external to the process itself. These resource descriptors are allocated independently and are sharable across processes (Figure 2).

The resource descriptors themselves include a reference counter. Each process that attaches to a descriptor increments its reference count. Conversely, each process that terminates or detaches from a resource decrements its reference count. The descriptor persists only so long as its reference count does not reach zero. New resource descriptors are allocated dynamically from the UMAX 4.2 kernel's non-paged memory pool.

The classes of resource descriptors currently implemented in nUNIX are the six categories grouped in the USER structure: address space, file descriptors, signal management, usage control, priorities and protections, and process administration statistics.

Control of these independent resource descriptors is quite straightforward. The **fork()** system call causes the creation of new process data structures and new resource descriptors referenced by them (which are copied from the parent's resources). Very little extra overhead is implied by the addition of another level of indirection, which in any event lends itself to being optimized away by the compiler. The existence of independent resource descriptors is invisible not only to user-level applications, but to much of the kernel itself. Alternatively, two new system calls provide for customization of the resource set associated with a process.

A new interface called **sfork()** creates the equivalent of a kernel-level thread. The actual implementation falls out of the restructuring of resource descriptors. **sfork()** is merely a **fork()** in which the resource pointers of the parent's process data structure are copied and the reference count of the individual resources incremented instead of the resources themselves being copied and initialized. The resulting thread of control is extremely lightweight but remains a UNIX "process," indistinguishable from any other. All existing system calls including **wait()**, **exit()**, etc. work as usual. **fork()** and **sfork()** may be freely mixed with no incompatibilities.

A second system interface called **resctl()** implements direct control over resource descriptors. **resctl()** is used to detach from an existing descriptor and to create a new one as well as to implement resource controls not supported by the existing UNIX process interface. All existing UNIX system calls that affect process resources work as usual with the qualification that all processes sharing that particular resource descriptor are impacted similarly.

Using the nUNIX interface, arbitrarily complex combinations of private and shared resources may be constructed by processes executing the same program. Suppose, for example, that two separate sets of file descriptors are desired among a number of processes which otherwise wish to share their environment. **sfork()** is used to create the processes while **resctl()** creates and attaches the second file descriptor resource. The cost of this configuration is increased only by the expense of creating a second array of file descriptors and the overhead of the system call itself.

The new system calls are:

sfork()
: Create a new execution environment that includes registers, program counter, scheduling data and pid. Attach this process to the current resources of the parent.

resctl()
: General control function for system resource descriptors.

A higher-level library interface to **resctl()** is also provided:

rcreate()
: Create and initialize a new resource descriptor of specified type. Attach current process to this resource and detach the previous resource.

rcopy()
: Create a new resource descriptor of specified type. Copy the state of the previously attached resource of this type. Attach current process to this resource and detach the previous resource.

This handful of additional system calls is quite flexible. The existing process interface may, for example, be defined in terms of the new primitives. The following pseudo-code segment describes the implementation of **fork()** in terms of **sfork()**, **rcopy()**, and **rcreate()**. All resources are copied from the parent, except for usage statistics.

```
fork()
{
  int pid;

  if ((pid = sfork()) == 0) {
    /* I am the child */
    rcopy(SF_MEM);
    rcopy(SF_IO);
    rcopy(SF_SIG);
    rcopy(SF_PROT);
    rcopy(SF_UCTL);
    rcreate(SF_STAT);
  }
  /* Both parent and child come here. */
  return(pid);
}
```

Preliminary Experience

The nUNIX kernel is still in its early stages and is intended mainly as a platform for experimentation with variable-weight processes and independent resources. Many limitations and unresolved issues remain.

The **sfork()** primitive provides an example of these issues. In order to maintain compatibility with **fork()** and allow a thread of control to return back up the call stack after an **sfork()** call, the parent's stack is copied into the new stack area of the child. For this scheme to work, pointers into the stack cannot be absolute references but must remain relative to the stack pointer. Currently, this is enforced only by convention. For the future, the decision must be made whether to enforce this limitation through compiler support or to disallow returning up the call stack, thus sacrificing orthogonality. Other, lesser issues of this type also remain.

Furthermore, nUNIX's resource sets need work. The existing partition was adapted merely as a convenience, because it corresponded to to the implied partitioning of resources in the current process data structures. Many changes to the existing design are contemplated including separate memory management and memory protection resources, the independent partition of process timers, and others. Thought has even been given to new classes of resources such as symbolic information and efficient communication areas shared between user and kernel. In addition, consideration will be given to making these resource pageable.

Semantic ambiguities such as those encountered in MACH [11,6], also remain. For example, what does it mean to send a signal to a process which shares a signal management resource? Does only that process receive it, or is it broadcast to all other processors sharing that resource, or is the signal taken by only the first available process which shares that resource? nUNIX has no limits on processes or resources to prevent any of the above strategies from being the default, but which is the "correct" approach?

Finally, the exact characteristics of the kernel are difficult to quantify without resolving many of the issues discussed above. In future publications, we hope to report timing data on the new constructs.

Despite these limitations, preliminary experience with the nUNIX primitives has been very encouraging. nUNIX supports a range of programming models and conforms to existing UNIX semantics. The system has proven itself to be particularly suited to an incremental implementation.

One unexpected result of nUNIX was that shared resources also appear to enhance the cooperating processes paradigm. In one example, almost 3 dozen NFS daemons simultaneously execute on our system. Because these daemons are stateless, they conform to the classic process model. It has proven useful, however, to share process statistics across them. A modified version of the utility **ps** shows statistics resources rather than pids and thus makes process management more rational. Instead of 34 entries for NFS daemons, we now see one.

nUNIX supports lightweight threads and UNIX processes within the same paradigm. Further, nUNIX processes may share some resources and have others private: the additional cost is paid only for those resources which are private in a "pay for what you use" strategy. Thus we have efficient threads for parallel programming, and still lose none of the strengths of conventional processes. This is a significant result that we hope can extend the life of UNIX.

A prototype kernel is being implemented on Multimax^{TM}, a shared-memory multiprocessor running both MACH, a new 4.3 BSD compatible kernel, and UMAX 4.2 a conventional (parallelized) adaptation of 4.2 BSD, upon which our prototype is based.

Conclusion

UNIX can accommodate the necessary support for parallel processing without requiring drastic changes to the fundamental notion of process. Rather than completely divorce the thread of program control from the set of program resources, our solution permits variable-weight processes, spanning the spectrum from multiple threads of program control executing within the same resource space (equivalent to the MACH notion of "threads" grouped into a "task") to threads sharing a subset

of their resources (such as file descriptors) to multiple threads executing with completely disjoint resource sets (equivalent to the "traditional" notion of a UNIX process).

The benefits of this approach are several:

- the notion of resource becomes orthogonal to the notion of thread execution,
- the kernel can support a range of processes, from a large number of very lightweight processes to a much smaller number of "traditional" processes,
- user-level applications and thread packages become significantly easier to implement,
- the user interface is uniform across process of all weights,
- the implementation is simple and clean.

Making efficient use of shared-memory parallel processors does not necessarily require the use of radically new operating systems or kernel interfaces. The fundamental UNIX concepts of process and resource need only be generalized. nUNIX represents one attempt at such a generalization and underscores yet again the remarkable adaptability of the UNIX Operating System.

References

[1] *Balance Guide to Parallel Programming.* Sequent Inc., Beaverton Oregon, 1986.

[2] *UMAX 4.2 Programmer's Reference Manual.* Encore Computer Corporation, Marlborough, MA 01581-4003.

[3] Z. Aral and I. Gertner. Non-intrusive and interactive profiling in Parasight. In *ACM/SIGPLAN PPEALS 1988 — Symposium on Parallel Programming: Experience with Applications, Languages and Systems*, New Haven, Connecticut, July 1988.

[4] J. Barton and J. Wagner. Beyond threads: resource sharing in UNIX. In *Winter 1988 Usenix Conference Proceedings*, February 1988.

[5] T. Doeppner. *Threads - A System for the Support of Concurrent Programming.* Computer Science Technical Report CS-87-11, Brown University, June 1987.

[6] T. Doeppner. *A Threads Tutorial.* Technical Report CS-87-06, Brown University, March 1987.

[7] T. Doeppner and D. Johnson. A multi-threaded debugger, extended abstract. In *ACM Workshop on Parallel and Distributed Debugging*, University of Wisconsin-Madison, May 1988.

[8] G. Hamilton and D. Code. An experimental symmetric multiprocessor Ultrix kernel. In *Proceedings of the Winter USENIX Conference*, 1988.

[9] I. Nassi. Encore's Parallel Ada Implementation. 1988. Private Communications.

[10] I. Nassi. A preliminary report on the Ultramax: a massively parallel shared memory multiprocessor. In *DARPA Workshop on Parallel Architectures for Mathematical and Scientific Computing*, July 1987.

[11] R. Rashid. Threads of a new system. *Unix Review*, August 1986.

[12] D. Ritchie. Computer Science Seminar given at Carnegie-Mellon University, 1988.

[13] A. Tevanian, Jr., R. Rashid, M. Young, D. Golub, M. Thompson, W. Bolosky, and R. Sanzi. Mach threads and the unix kernel: the battle for control. In *USENIX Conference Proceedings Summer 1987*, June 1987.

[14] A. Tevanian, Jr., R. Rashid, M. Young, D. Golub, M. Thompson, W. Bolosky, and R. Sanzi. A UNIX interface for shared memory and memory mapped files under Mach. In *USENIX Conference Proceedings Summer 1987*, June 1987.

System V/MLS Labeling and Mandatory Policy Alternatives

Charles W. Flink II
Jonathan D. Weiss

AT&T Bell Laboratories

INTRODUCTION

System V/MLS is a product under development by AT&T to meet customer requirements for a National Computer Security Center (NCSC) certifiable operating system.[1] A key design goal for System V/MLS is B level certifiability while maintaining SVID compatibility.[2] The intent behind this effort is to introduce enhanced security, while preserving to the maximum possible extent those features of the UNIX operating system that have made it successful to date.

Certifiability at the B levels requires the introduction of a class of associations and restrictions that have not existed previously in the UNIX System. It is necessary to associate security labels with all subjects (users, processes) and objects (files, directories, devices, interprocess communication channels, etc) and provide a Mandatory Access Control (MAC) mechanism using the labels to implement a security policy strictly limiting the flow of classified data to only those subjects "cleared" for access to the data. The concepts of labeling and mandatory access control (MAC) are new to the UNIX environment, and therefore present a number of new technical challenges. These challenges are multiplied when it is necessary to implement the security labels and associated mandatory controls in a manner that preserves a simple interface, and maintains compatibility with UNIX System V.

This paper will discuss design alternatives and decisions with respect to labeling and mandatory policy for the System V/MLS system. Our criteria in evaluating the alternatives are:

— certifiability,

---

1. The product described in this paper is not to be interpreted as a new *standard* release of AT&T System V, nor should it be considered a statement of how the *System V Interface Definition* (SVID) will necessarily evolve to address NCSC security requirements. The product is available from the AT&T Federal Systems Division (FSD) as an *enhanced security* version of System V Release 3. It is being sold to government customers and vendors under the understanding that it cannot be guaranteed to be fully "upward compatible" with as yet unspecified future AT&T, national, and international standards for secure systems. It will be sold and supported by FSD until standards are established and AT&T System V evolves to meet the special security requirements of the government systems marketplace.

2. Release 1.0 of the product meets this requirement as defined by *System V Interface Definition, Issue 2*, as tested by the *System V Verification Suite, Release 3* (BA/OS, BA/LIB, BA/ENV, KE/ENV, and KE/OS).

— compatibility,

— simplicity of use and interface,

— flexibility.

Design issues associated with other elements of the System V/MLS system will be reserved for future discussion.

LABELING

Various options were considered for the storage, representation, and association of subject and object labels. Assessments of customer requirements indicated that our labeling approach needed to be flexible in order to handle a number of specialized environments. In particular, we concluded that the NCSC guideline that labels should be capable of representing 16 hierarchical levels and 64 categories is insufficient for certain environments. Furthermore, we observed that some environments require labeling mechanisms that vary greatly from the notion of hierarchical levels and categories. Thus, a primary consideration in the development of our labeling scheme was flexibility. Our decisions with respect to label storage, representation, association, and handling were designed to reflect this requirement.

Whenever possible, characteristics of the label has been parameterized, functions have been provided to hide actual label representations from applications, and the kernel-level code implementing labels and mandatory policy has been packaged in a separate module to simplify tailoring of labels to fit new application environments.

1. Label Storage

In the interest of compatibility, and to expedite porting of an identified set of applications, we were determined to implement labeling in the UNIX System without modifying any of the underlying data structures. This approach allowed us to maintain not only an upwardly compatible system call interface consistent with the *System V Interface Definition* (SVID), but also a high degree of compatibility and interoperability with other interfaces: protocol implementations, device drivers, established support procedures and organizations, conventional UNIX systems run in the System-High mode, and conventional user training, practice and experience. Given current market pressures, it is *not* reasonable in our opinion to establish an entirely new flavor of UNIX systems with incompatible kernel-level interfaces, incompatible backup/restore and filesystem structures, system programming and support personnel organizations, etc.[3]

Given the desire to add labels without modifying system data structures, it was necessary to identify an existing field in each applicable system data structure that could be used to contain label information without introducing gross incompatibilities. The group identifier field (GID) has been chosen as a likely candidate based on the similarity of its characteristics to those of labels. In fact, it could be argued that the group identifier is the "natural" label for UNIX objects and subjects.

3. A departure from the existing internal structure of the UNIX system *will* be necessary for the long-term evolution of the UNIX product to address NCSC requirements at higher levels. Certainly, major internal structural changes are required at B3 and beyond. But such restructuring will likely obsolete most of the growing pool of device drivers and protocol modules and should not be undertaken lightly.

For example, there is a GID field for every subject as part of the process table entry and for every object as part of the inode, IPC data structures, etc. Every object is "stamped" with the effective group identifier of the subject process that creates it. Child processes inherit the group identifier of their parent process. The system implements a policy that provides basic protection for this group "label" (though this policy needed to be made mandatory rather than discretionary.) Further, this group label is maintained by archiving utilities, such as *tar*(1) and *cpio*(1) and displayed by utilities such as *ls*(1) and *id*(1). Groups are already currently used in UNIX systems run in system-high mode to separate various projects, often directly or indirectly representing project sensitivity. It is on the basis of these label-like characteristics of groups that we decided to extend them to implement clearances and classifications.

2. Label Representation

Once the conclusion was made to employ the group identifier for label storage, the question became "how can the sixteen bits (or 65,536 unique GIDs) be optimally used to represent security labels, *as well as* UNIX discretionary access control groups?"

2.1 Direct Labeling

A preliminary approach taken in the System V/MLS alpha release was to divide the group ID as follows:

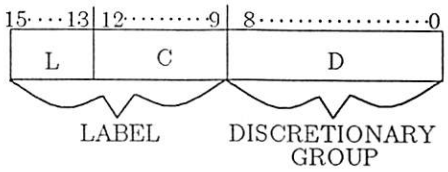


Figure 1. DIRECT LABELING

Such a division allowed us to represent up to eight different hierarchical levels (L), four different categories and combinations thereof (C), and 512 different discretionary groups (D).[4]

The advantages to this labeling approach are:

- simplicity;
- minimal impact on standard System V
  - — user interface,
  - — system calls,
  - — data structures,
  - — kernel code;
- "natural" use of UNIX grouping.

4. The SVID dictates that 100 of these must be reserved for administrative groups, leaving 412 for users.

On the negative side, however, this approach is limited with respect to:

- number of levels;
- number of categories;
- number of discretionary groups;
- extensibility (e.g., support of new fields);
- flexibility.

The System V/MLS alpha release used this labeling scheme for the purpose of demonstrating System V/MLS characteristics and interfaces to selected customer sites and to provide a base for experimental porting of applications. A library of label interface routines has been provided to hide the actual label representation in order to facilitate migration to the more flexible, more general solution: indirect labeling.

2.2 Indirect Labeling

Customer requests for greater flexibility and more levels, categories, and discretionary groups prompted us to adopt an indirect labeling approach for our product. In this approach, the GID field (group identifier) no longer directly contains a label and discretionary group, but instead indexes into a data structure containing this information. Starting with Release 1.0, all System V/MLS releases use indirect labels.

2.2.1 Extensibility The GID-referenced data structure is not necessarily restricted to just security label and discretionary group; it may also contain a variety of other, less common protection mechanisms (e.g., discretionary caveats (handling instructions); special "privileges" granted to subjects marked with the GID; label ranges for multilevel devices or directories; group mandatory/discretionary access control lists, etc.) See figure 2 below.

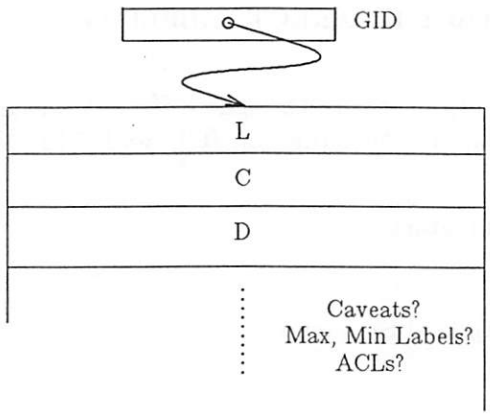


Figure 2. INDIRECT SECURITY LABEL

The advantages of the indirect GID-based labels are:

- minimal impact on standard System V
  — user interface,

— system calls,
— data structures,
— kernel code;

- "natural" use of the UNIX system group concept;
- many levels, categories, and discretionary groups;
- extensibility (e.g., space for "privileges", etc.)
- flexibility for specialized policies.

The disadvantages are:

- limited number (65,536) of GIDs available;
- performance/complexity involved in accessing label from the kernel level.

Neither of these disadvantages have proved to be significant. The performance concern warranted a memory caching scheme for the label data structure, but this was a minor effort compared to alternatives such as new filesystem structures, etc.

2.2.2 GID Range Limitations Since the new data structure presents no limits on the size of the various fields, any number of levels, categories, and/or discretionary groups may be supported. There is the limitation, however, that the system may not have any more than 65,536 unique protections/authorizations ("groupings"). Early analysis indicated that it would be highly unlikely that any one system would approach this limit. Objects are naturally grouped by users to be manageable. For example, most files for a given project will share the same group designation. As a result, far fewer groups are needed than objects themselves. Few UNIX systems contain as many as 65,000 files, let alone *groups* of files. We expect that the typical system would use less than 1000 unique labels.[5]

2.2.3 Performance Impact Access checks performed with indirect labeling often involve just a check for identity between the GID values associated with the subject and object. This is because most objects are accessed by the subject (process) from which their protections were inherited (at their creation) or subjects working on the object and hence likely in the same group at the same classification level. For example, a subject (process) typically creates and then accesses a set of temporary files with protections based directly on the subject's (process') authorizations. For such identity comparisons, there is no need to additionally and separately perform the mandatory access control checks. Similar "short circuit" evaluations occur at several points within the access control evaluation. The result is very little overhead for the average access and very infrequent "cache misses" in the labels cache. Label overhead is estimated to be 1% for typical installations, but is highly dependent on the number of distinct labels used and the distribution of label usage among users.

---

5. To handle label translation for objects moved between systems with differing label definitions, the same tools can be used as are currently used to handle the movement of files between systems having different group designations. As a result, the total universe of labels for any fixed set of meanings assigned to category bits and hierarchical levels is *very* large. The 60,000 unique label limit only applies to any single machine.

2.2.4 Assurance Issues The use of indirect labels and the problem of managing the association between GID and label seem to raise some assurance concerns. These have been carefully addressed in the design of the TCB. First, the label to GID mapping is immutable. Once a label associated with a GID (when the GID is allocated via *mkgrp* or *mkpriv* commands), neither the label nor the associated GID can be changed by any activity short of intervention by trusted maintenance personnel. Further, a GID cannot be reused (assigned a new label) until an administrator specified interval of time (e.g. 1 year) has passed since any object marked with the GID has existed in the system. If a GID is to be retired, all objects marked with the GID must be removed or reassigned before this "clock" starts. Secondly, the default label is "system high". If an object is imported or otherwise introduced to the system with a GID not mapped in the label file, the system grants no access to the object until the object is assigned a known GID (and hence label) by a trusted administrator.

An initial concern raised during the evaluation of indirect labels for the System V/MLS product was that the association between a GID and a particular label is host environment dependent, and thus, insufficient as an exportable label. Note, however, that labels with explicit numbers for levels, and bits for categories (i.e., the direct approach described above) are also environment-specific with respect to exportation. The particular numerical levels and individual bits only have meaning in the specific environment in which their mappings are defined. For instance, the bit corresponding to a particular category can mean "NATO" on one machine and "NOFORN" on another. Even if the bit represents the "same" concept across machines (e.g., NOFORN - **no for**eign **n**ationals), the actual meaning of the bit can be different depending on the nation in which the system is operated. As a result, whenever *any* label representation is exported, it must be a procedural requirement that its meanings (or mappings) be exported as well. Thus, the indirect label approach suffers no added risks when compared with direct labeling.

3. Label Handling

3.1 MLS Module

Through the use of indirection and a label interface library, the actual label representation for System V/MLS is kept hidden and isolated. We further isolate the interpretation of (and operations performed on) labels in a kernel level module known as the MLS module.

The MLS module is a removable and replaceable portion of our system that allows us to easily adapt and evolve our labeling scheme and security policy to track our customers' needs. It is implemented through the insertion of "hooks" in various UNIX kernel routines that call the appropriate functions in the MLS module when policy decisions must be made. A similar structure has been established for the System V/MLS security audit trail (SAT module). This modularity is discussed in system design documentation.

The following diagram illustrates the interaction between a UNIX kernel routine, and functions within the MLS (and SAT) modules. The example below (figure 3) shows the calling sequence for enforcement of access controls (and access auditing) for the *read*(2) resulting from the "cat" command.

The label representations, and the rules applied, are not visible in the kernel routines. They are solely the concern of the MLS module.

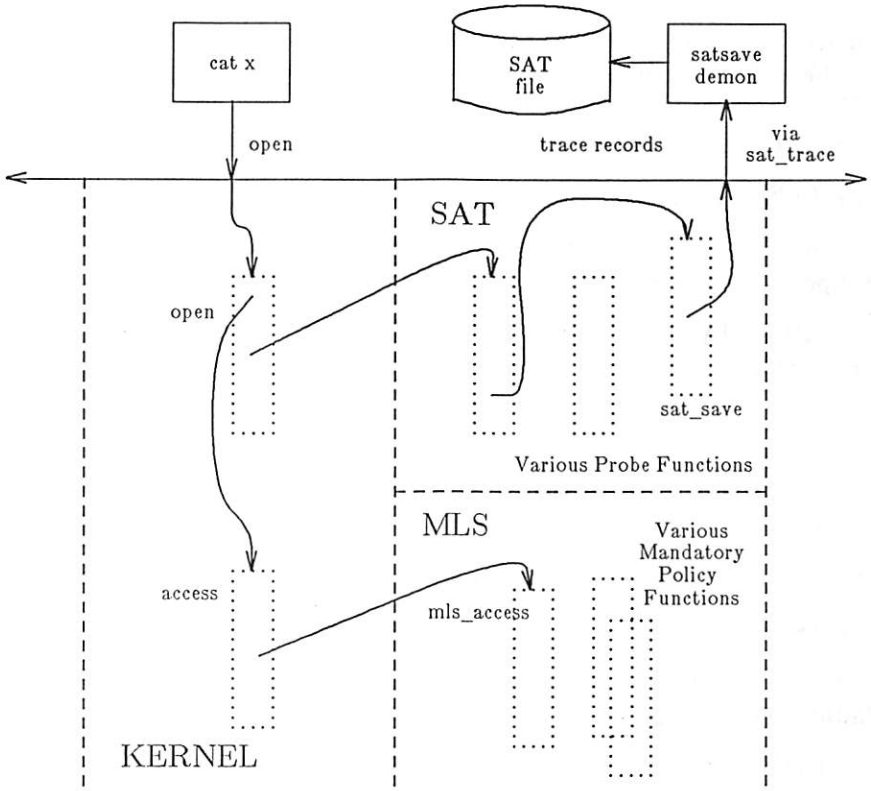


Figure 3. System V/MLS MODULES

3.2 Projects Interface

The structure of System V/MLS supports an enhanced version of the standard UNIX system discretionary access controls. Although this is primarily a paper on labeling and **mandatory** policy, it is important to briefly discuss the connection between System V/MLS labels and its discretionary access control (DAC) mechanism.

System V/MLS has a "project" oriented DAC interface that allows selected individuals to set up and manage projects. A "project" is simply a discretionary group in the conventional UNIX sense. Because it may be defined at multiple specified clearance and classification levels, there may be more than one GID and group file entry for any given project. For example, a project "shuttle" can be defined in System V/MLS, and associated with permissible clearances/classifications (e.g., topsecret, or secret-nato, or confidential-nato-crypto). With such associations established, subjects and objects may only operate/exist within the project if they are cleared/classified at one of the allowable levels. Likewise, subjects' and objects' clearances/classifications may only be changed to an allowable level within their current project.

Membership of users in the various clearance levels of a project is controlled by the project administrator. (Simply the "owner" (first member) of the group.) A project administrator can grant or deny membership in a project (consistent with user clearances) at the user level of granularity. An *object owner* may further decide to grant read, write, and/or execute access to him/herself, all users within the object's current project, and/or everybody, by setting the proper UNIX system protection bits (via *chmod*(1)). Thus, one can think of projects as shared access control lists,

administered by project managers, that can be switched on and off by object owners. If an existing project does not provide the proper discretionary protections for a file, a new one may be easily created by the project administrator or another authorized individual. The intervention of the system administrator (or broad granting of superuser privilege) is not required.

MANDATORY POLICY ALTERNATIVES

Prior to performing any discretionary checks, the System V/MLS mandatory access control (MAC) policy of the System V/MLS system is applied.

The System V/MLS MAC policy was the result of analysis of various alternatives for mandatory protection of:

- files,
- inodes,
- directories,
- IPC mechanisms (and their associated data structures),
- and processes (as recipients of signals).

Each of the alternatives attempted to map the simple security (ss-) and *-properties of the Bell-LaPadula security model into a UNIX framework.

The following table shows the UNIX system operations involved in enforcing the mandatory controls; it also defines the notation used in comparing alternative policies:

Symbol	Represents
R	Read (file, directory, etc.)
S	Search (directory)
E	Execute
W	Write (overwrite or append)
O	Overwrite
A	Append
C	Create (creat(2), mkdir(2), mknod(2), etc.)
L	Link (ln(1), link(2))
U	Unlink (rm(1), unlink(2), or rmdir(2))
St	Read file/inode status (stat(2), etc.)
Ch	Change status (chown(2), chmod(2), utime(2), etc.)
K	Send a signal (kill(2))
Ripc	Read IPC mechanism
Wipc	Write (Alter) IPC mechanism
O	the Object in question
Od	Directory Object or Directory containing Object
Of	simple File Object (not directory)
S	Subject (process acting on user's behalf)
>=	label of left item "Dominates" label of right
==	Identical security label

Figure 4. MANDATORY POLICY NOTATION

For IPC mechanisms, the Ripc and Wipc symbols in the above notation represent a class of operations. All IPC operations can be mapped into one of these two classes (see the *UNIX Programmers' Manual*). The Od and Of symbols are used to distinguish between checks on a file-type object versus its parent directory. The meanings of the other symbols will be explained in more detail as they are encountered.

1. File Protections

At first glance, neither the "no read up" (ss-property) nor the "no write down" (*-property) rules seemed to place restrictions on writing *up* for files. We therefore derived the following policy for file operations:

Operations	Dominance Relation
R/E	S >= O
W(O/A)	O >= S

Figure 5. FILE PROTECTION POLICY 1

The primary problem that we perceived with this policy is that it violated a "no destroy up" principle that, although not a requirement in the "Orange Book", seemed to us to be fundamental for data integrity. We therefore modified our file protection policy as follows:

Operations	Dominance Relation
R/E	S >= O
O	S == O
A	O >= S

Figure 6. FILE PROTECTION POLICY 2

This solved the "no destroy up" problem, but left a covert channel (*-property violation) in the form of the error codes returned when an attempt is made to append to a file of strictly dominating classification. For example, a high-level user can turn on and off the discretionary write permissions on a high-level file, thereby signaling a low-level user making multiple attempts to append to the file. This problem can be solved by implementing "blind" appends (success code always returned), but we deemed this an unreasonable and undesirable mechanism for real-world systems. We therefore arrived at the following policy:

Operations	Dominance Relation
R/E	S >= O
W(O/A)	S == O

Figure 7. FILE PROTECTION POLICY 3 (SELECTED)

Since this policy no longer allows "write up" in any form, the covert channel discussed above is not an issue. This alternative is more restrictive than the others, and forces upward communication to be accomplished through other means (e.g., a higher level process reading "down" rather than a low-level process writing "up".) For integrity

reasons, we decided it was the superior approach, as it prevents low-level users from affecting objects at higher levels.

2. Inode Protections

Having obtained a reasonable policy for protecting file information, we moved on to the inodes. We noticed that a parallel existed between operations that queried information in the inode and file reads, and operations that altered information in the inode and file writes. This led us to the following set of rules:

Operations	Dominance Relation
St	S >= O
Ch/L/U	S == O
U (last link)	S >= O

Figure 8. INODE PROTECTION POLICY 1

Here, stat (St) and change (Ch) are the primary inode observation and alteration functions, and link (L) and unlink (U) are included because they change the link count field of the inode.

This policy seemed reasonable and allowed no obvious covert channels[6], but was inconsistent with the current structure of standard UNIX system protections: in the UNIX system now, directory permissions are used to determine link and unlink capabilities for objects within them. We concluded that it would be perfectly valid for the label on the parent directory to be the basis for the controls on link and unlink operations. Applying the parent directory label for link and unlink, and the object label itself for stat controls, can create a covert channel only when the object is classified at a lower level than the parent. This may only happen in System V/MLS when an object is declassified by a security administrator (or locally authorized individual). We therefore chose to make prevention of the covert channel a procedural issue for declassification, rather than an inode access protection issue. The following policy resulted:

Operations	Dominance Relation
St	S >= O
Ch	S == O

Figure 9. INODE PROTECTION POLICY 2 (SELECTED)

6. There is the matter of access times in lower-level classified objects being updated by reads by higher level subjects. This is a classic example of a covert channel. The ultimate solution will require a redefinition of the access time feature: either don't update the access time on reads from higher level subjects or don't report it via the *stat*(2) system call. The expedient alternative is to document this as a covert channel and throttle as necessary. E.g.: *stat*'s after reads from above will be delayed X seconds.

3. Directory Protections

Having decided to use parent directory labels for create, link, and unlink controls, we considered the following policy for directory protections:

Operations	Dominance Relation
R	S >= O
S	(no mandatory check)
C/L	Od >= S
U	Od >= S and S >= Of

Figure 10. DIRECTORY PROTECTION POLICY 1

This policy was attractive for its flexibility and partial resolution of a class of problems related to temporary directories. The UNIX System relies on a number of directories (/tmp, spool directories, etc.) that, with the introduction of labeling, become storage places for objects with multiple classifications. In conventional UNIX systems, temporary directories represent a problem because the discretionary rule that allows any subject to create a file in the directory also allows any other subject to remove the file. With the introduction of labels, there is the extended problem of establishing rules to allow files of numerous levels in a single directory such that they may still be appropriately protected. Under the "Directory Protection Policy 1" (figure 10 above), such directories would be labeled system high and files of all levels could be stored in them. The rule for unlink would assure that no higher level information could be removed by a lower level user, and the rule for search would permit users to access the objects for which they were authorized in the higher level directory.

The problem, however, was that the search rule allows lower level users to test for the existence of higher level objects. Even though the rule for directory read prevented the lower level subject from listing a directory, if the subject knew the name of the object, it could attempt to access the object and note success or failure. This situation represented a covert channel. The problem may be partially corrected by making the return code for failure to satisfy the mandatory controls the same as for object not found, but it would still be possible to check for the existence of classified directories by attempting to search them. The bandwidth on this inherent covert channel could be reduced, but we decided that our customer base is less interested in the flexibility provided in this approach than in the additional security provided by the following:

Operations	Dominance Relation
R/S	S >= O
C/L/U	S == Od

Figure 11. DIRECTORY PROTECTIONS 2 (SELECTED)

This policy is certainly stricter than the first. Objects may only be created, linked to, and unlinked from directories with labels identical to that of the subject. Furthermore, the directory's contents may only be detected by subjects whose labels dominate. The only means by which an object may have a label that differs from that of the parent directory is if the object is created and then reclassified in an authorized manner (for System V/MLS, the rule is that all subjects can upwardly reclassify objects that they

own, while designated security administrators can declassify).

This policy does encounter the problem of temporary directories, however. Because of the identity rule for create/link, there is no way to classify directories that will permit them, under the stated policy, to allow creation of objects of multiple-level. This means that without some special corrective measure, all current applications that use temporary directories would no longer run.

For the alpha release of System V/MLS, we modified the policy as follows until such time as a solution to the temporary directory problem could be selected:

Operations	Dominance Relation
R	S >= O
C/L/U	S >= Od

Figure 12. DIRECTORY PROTECTIONS 2 (MODIFIED - ALPHA RELEASE)

This policy allowed a covert channel in the form of ability to create higher-level classified file/directory names in lower-level directories and was thus unacceptable in the long term. See section 5 for a discussion of alternate solutions to the problem of shared (e.g. /tmp) directories.

4. IPC and Signals

The policies for IPC mechanisms and signals are straightforward[7]:

Operations	Dominance Relation
Ripc	S >= O
Wipc	S == O
K	S == O

Figure 13. SIGNAL AND IPC PROTECTIONS (SELECTED)

The only issue in these policies was whether signals may or may not be sent to processes with labels dominating the subject's label. We decided that it was better to protect the higher level processes from being affected by the lower level ones.

5. 'tmp directory' Alternatives

A number of solutions for the temporary directory problem have been proposed. Key solutions considered were:

- label ranges for directories,

7. There is a high-bandwidth "covert channel" issue that results from the fact that IPC objects share a common unprotected name space. A Trojan horse operating at a classified level can signal an accomplice by creating or deleting IPC objects in the common name space. Several alternative solutions for this problem are under consideration.

- multi-level "subjective" directories,
- cloning devices,
- parametrized symbolic links.

5.1 Label Ranges

One means for solving the problem is to apply similar controls to directories as to multilevel devices, i.e. to have a maximum and a minimum label associated with the directory. This allows a range of classified objects to be placed in the directory, with removal controls applied based on the object label itself. This solution is not consistent with the selected mandatory policy, or the current philosophy of UNIX system access checks, and was therefore rejected in System V/MLS.

5.2 Subjective Directories (SELECTED)

Another solution is to alter the *namei()* routine to recognize special directories flagged as "subjective". For these special directories, *namei()* can be modified to insert the subject's current security label (suitably encoded) into the target path, invisibly changing all directory references to refer to a suitably labeled (and hence protected) subdirectory.

This proposed solution is called "subjective directories" because the directory given to the user is not the one strictly identified by the path. It is instead a different directory selected by the security level of the subject. The "objective" or "true" view of the file tree is available only to administrative processes. This mechanism has not only been applied to the /tmp directory problem, but also to directories like /usr/mail and /usr/spool/lp to implement multilevel mail and lp with limited changes to these subsystems. The /dev directory as also been handled this way, hiding from the user the vast majority of the devices usually defined on a system. Only those devices are visible that are available to the user (i.e. classified at the user's current operating level and designated as user accessable by a trusted application.)

Directories can be recognized by *namei()* as being subjective by applying this meaning to the currently unused directory "Set-GID" bit. Likewise, it may be desirable to establish subjective directories parameterized by UID as well, using the "Set-UID" bit as a flag.

System V/MLS uses the "subjective directory" approach to the /tmp directory problem. However, instead of appropriating the directory "Set-ID" bits, which might be applied to some other purpose in future UNIX System V releases, it flags the directory with the special group "SECURED". The *namei()* function, upon encountering a directory with this special group, performs the substitution described above.

5.3 Cloning Devices

An alternate approach to solving the temporary directory problem can be based on a pseudo device or new file system type that "clones" a private, appropriately labeled, secure directory for every appropriate new access to the actual directory. The new file system or cloning device can be mounted over the /tmp, /usr/tmp, and any other directory for which "subjective directory"-like functionality would be desired. Such a solution requires additional effort to implement, and no advantages over the subjective directory approach.

5.4 Parameterized Symbolic Links

As a final alternative, the needed functionality can be achieved by an extension of the symbolic link concept. The symbolic link of Version 8 or 4.2 BSD is simply an absolute path which is substituted for the path being interpreted by *namei()*. If the symbolic link is extended to include variables in *sh*(1) notation, which will then be expanded by *namei()* from the environment of the process (*environ*(5)), a highly flexible solution results. The **/tmp** and **/usr/tmp** files could be symbolic links to /tmp.d/$LEVEL and /usr/tmp.d/$LEVEL, where LEVEL is an environment variable set upon entering a classified level. The LEVEL variable is then set to the classification level just entered. As a result, the symbolic link can be made to point to an appropriately classified subdirectory, segregating temporary files based on classification and thereby protecting them.[8]

Further, this technique can be generalized to any other example of a shared resource directory. For example, *mail*(1) can be easily used as a multi-level secure mail system if the directory **/usr/mail** is in fact a parameterized symbolic link to **/usr/securemail/$LEVEL**. Invocation of mail from a classified level will automatically access a suitably labeled classified mailbox corresponding to the level in question. Similar examples can be provided for *lp*(1), **/dev/tty**, and **/dev/null** simplifying the implementation of mandatory protection for these resources as well. If and when symbolic links are included in future releases of System V, this may become the best alternative for solving the tmp directory problem.

6. Policy Summary

To summarize, our full selected policy is as follows:

Operations	Dominance Relation
R/S/E	S >= O
W(O/A)	S == O
C/L/U	S == Od
St	S >= O
Ch	S == O
Ripc	S >= O
Wipc	S == O
K	S == O

Figure 14. System V/MLS MANDATORY POLICY

We feel it provides the proper combination of flexibility and security for the various access operations. Since the implementation of the mandatory policy is localized in a limited set of MLS module routines, it is evolvable to future requirements. Thus, it can

8. Note that it is not necessary to trust the environment variable. If some untrusted program or devious user changed LEVEL to lie about the true current operating level, the link would simply fail to take the user to a usable directory. Mandatory policy would still be in effect and would be computed without regard to the LEVEL variable. If the resulting directory was classified "higher" that the user, it would be totally inaccessable. If "lower", the user would be able to read the directory and probably read the files it contains, but would not be able to create files.

be made more flexible *or* more secure depending on customer needs.

There are a number of additional elements not discussed here, that influence the security policy of the system (e.g., level changing commands, declassification). They are described in the informal System V/MLS model.

CONCLUSION

We believe that the chosen labeling scheme and mandatory protection policy can be taken beyond class B1 to meet the requirements at the higher certification levels. In implementing labeling and mandatory policy without embedding assumptions about label format or meaning within the kernel itself, System V/MLS provides a degree of flexibility allowing implementations that can go well beyond standard "Orange Book" mandatory protection model. We intend to maintain the principles of flexibility, simplicity, and compatibility throughout the product's evolution.

CREDITS

The authors would like to thank the many individuals that have contributed to the System V/MLS effort over the years and have contributed directly or indirectly to this paper through their insights and ideas. This paper highlights only a portion of the work that has gone into the production of System V/MLS Release 1.0 and the planning, analysis, and prototyping work that continues to contribute to the rapid development of new features and interfaces to be available in subsequent releases.

REFERENCES

1. "DoD Trusted Computer Systems Evaluation Criteria," United States Department of Defense, No. 5200.28, December 1985.
2. L. K. Barker and L. D. Nelson, "Security Standards—Government and Commercial," *AT&T Technical Journal*, Vol. 67, No. 3, May/June 1988, pp. 00-00.
3. *System V Interface Definition*, Vol. 1 & 2, AT&T Customer Information Center, Indianapolis, Indiana, 1986.
4. *UNIX System V, Release 3 User Reference Manual*, AT&T Customer Information Center, Indianapolis, Indiana, 19xx.
5. D. E. Bell and L. J. LaPadula, *Secure Computer Systems: Unified Exposition and Multics Interpretation*, MTR-2997, Revision 1, MITRE Corporation, Bedford, Massachusetts, March 1976.
6. *UNIX System V, Release 3 Programmer's Manual*, AT&T Customer Information Center, Indianapolis, Indiana, 1988.

Secure Multi-Level Windowing in a B1 Certifiable Secure UNIX® Operating System

Barbara Smith-Thomas

AT&T Bell Laboratories

ABSTRACT

Until now multi-level secure operating systems have been designed to be connected only to so-called "dumb" terminals because we have assumed that intelligent terminals are necessarily insecure. An intelligent terminal does have considerable potential for compromising the security of the system; the more intelligent the terminal, the greater the hazard.

AT&T Federal Systems Division and AT&T Bell Laboratories are developing a B1 certifiable multi-level secure UNIX operating system, called System V/MLS. System V/MLS supports an intelligent terminal: the AT&T 630 Multi-Tasking Graphics terminal. With the System V/MLS security enhancements, the 630 MTG terminal supports trusted labeling of windows, monitoring of cut and paste to insure compliance with the SV/MLS Mandatory Access Control Policy, a mouse-based, menu-driven user interface, controlled local processing, and terminal sanitization on logout.

By designing and implementing appropriate controls we have achieved even more security for an intelligent terminal than is possible with a "dumb" terminal.

1. *INTRODUCTION*

Before describing the security modifications that have been implemented in System V/MLS to secure the intelligent terminal, it is appropriate to describe the terminal's base features. The AT&T 630 MTG intelligent terminal is a high resolution, multi-window, dot mapped, multi-host, programmable graphics terminal [AT&T87a]. It supports two terminal environments: layers and non-layers. In the *layers* environment, a terminal resident window manager supports up to seven concurrent, high resolution graphics terminal sessions over each of two host connections. The window manager supports cutting and pasting of text between windows, and between hosts. In the *non-layers* environment the

630 operates as a single high resolution graphics terminal. In either environment the terminal has the capability of running downloaded application programs. It permits caching of downloaded programs, and sharing of both program and data space between downloaded applications. The terminal also supports a printer connection.

System V/MLS is an AT&T Federal Systems special product being developed to address Department of Defense (DoD) computer system security requirements [FLIN88]. System V/MLS is a version of the UNIX System V operating system, modified to support Multi-Level Security features in compliance with criteria specified by the National Computer Security Center (NCSC) [DOD]. System V/MLS is designed to provide multi-level security at the B1 level. It has completed a year-long developmental evaluation and was submitted for formal evaluation in October of 1988.

Connection of an intelligent terminal, such as the 630 MTG terminal, to a multi-level secure operating system has the potential to compromise the security of the system. For example, cutting and pasting text from a window at a higher classification to a window with a lower classification is a declassification that may or may not be permitted, but which must be audited if it is permitted. The substantial local memory and processing capability of the terminal can reveal sensitive information after a user has terminated his or her host session, even after the original user has left the workstation.

In this paper the modifications to the 630 MTG functionality that were required to incorporate it into System V/MLS are described. The aim was to retain as much of the functionality as possible within the security restraints imposed by the "Orange Book"[DOD] requirements and the System V/MLS security policy. In section 2 the architecture of the UNIX layers/xt/630-terminal connection is described and the 630 MTG firmware modification mechanism is outlined. Section 3 lists the capabilities of the 630 MTG terminal that have been retained in System V/MLS along with some of the constraints that have been imposed on these capabilities to insure security. Section 4 contains a discussion of lessons learned, and section 5 outlines plans for enhancements to be added in the immediate future.

2. ARCHITECTURE

2.1 Logical and Physical Connections to the Host

The terminal/host connections are unchanged in the secure version of the 630 MTG terminal, except that the secure version only permits connection to one host at a time. A 630 terminal operating in *layers* mode has two to eight logical connections to the host computer multiplexed over a single tty connection. There is one communication channel for each active host window, plus a control channel. When invoked, a *layers* process selects eight xt pseudo-terminals, xtnm0 through xtnm7, where n and m are digits. The control channel is xtnm0; the data channels are xtnm1 to xtnm7. (In Figure 1 the "00th" set of xt

pseudo-devices are depicted: xt000 through xt007. Several such sets can be supported simultaneously on any SV/MLS host.)

When a window is created, a data channel is associated with it. The *xt driver* multiplexes input/output through the xt devices onto the real tty associated with the *layers* process that manages the host end of the windowing protocol. The shells, or other host resident application programs, can ignore the fact that they are executing in a *layers* environment; each *xt* device appears to be an ordinary terminal. (See Figure 1.)

On the 630 MTG side of the physical port a demultiplexer/controller, *demux*, receives the communications from the host. Data intended for a window is simply passed to *wproc*, or a downloaded function in control of the window. Control instructions, such as a download initiation signal, are handled by *demux*.

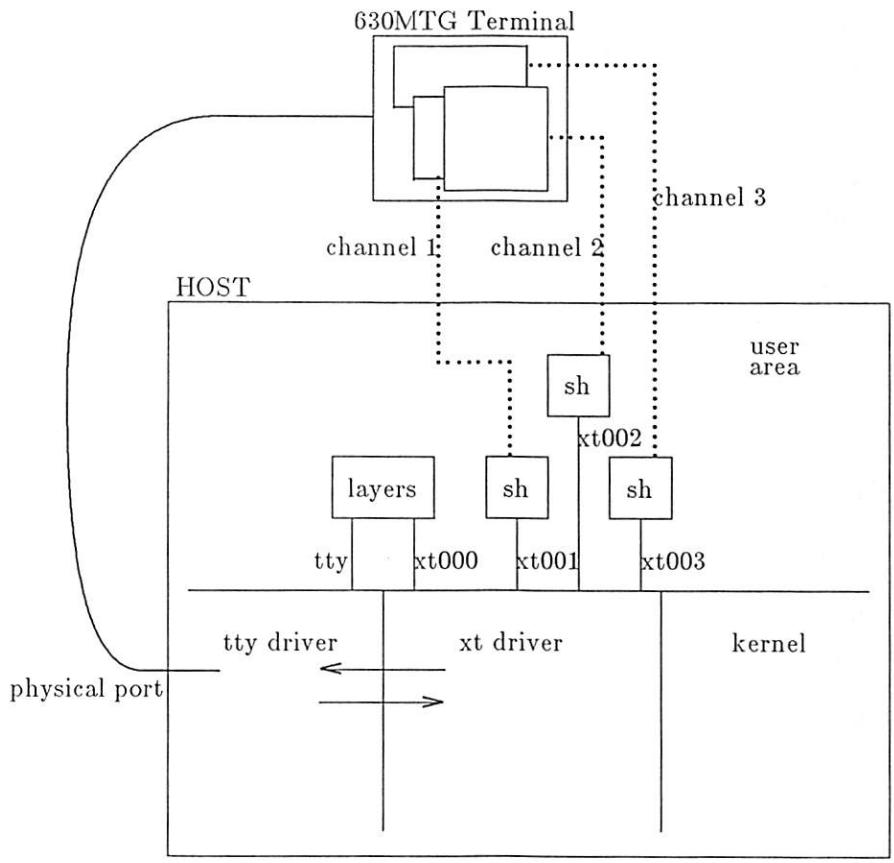


Figure 1. 630 MTG Communications Overview

2.2 The 630 MTG Firmware Modification Mechanism

The internal 630 MTG hardware and firmware architecture provides for a vector table containing firmware function addresses and static variables to reside at a fixed place in the terminal RAM. The terminal download function, *dmdld*,

supports a takeover download option, by means of which modifications to firmware functions can be loaded into known locations in RAM. These known locations can be entered into the vector table in place of the firmware addresses typically stored there. Invocation of firmware functions will be deflected to the downloaded functions. This mechanism makes it relatively convenient to modify the 630 MTG terminal firmware. The vector table mechanism also permits the downloaded firmware modifications to access the global variables maintained by the terminal firmware.

3. REQUIREMENTS

The following 630 MTG functions are retained in the SV/MLS environment, with restrictions and modifications as indicated by italics:

- Windowing — The 630 MTG supports up to seven windows directly connected to *a single* SV/MLS host. The 630 MTG supports local windows. The number of local windows is restricted only by the amount of terminal memory. A local window is created by making a copy of a host window. All information connected with the host window and any terminal processes running in that window are transfered to the local window. The host window is then reinitialized. There are two "system" windows: SETUP and PF-edit. The SETUP window contains settings for terminal configuration option. The PF-edit window is used for programming the Programmable Function keys.

- Buffering — Buffering is permitted in both host and local windows. In the 630 MTG terminal, 'buffering' refers to the management of scroll memory associated with each buffered window. The scroll memory or buffer is accessable via operations on the 'scroll bar' on the right side of a buffered window. Buffering is turned on or off via the SETUP window.

- Editing — Any host or local window may be edited using the current, mouse-based 630 MTG window editing functions. All windows support 'cut' and 'paste'; a host window also supports 'send' in which text is sent to the host, which then echos it to the window.

- Data Transfer between Windows — Data may be transfered from one window to another through the Global Save Buffer (GSB). Information highlighted during a local edit can be copied or cut from a window and saved in the GSB. The contents of the GSB can be inserted into a window using either 'paste' which is a purely local transfer, or 'send' which sends the text to the host. 'Send' is only available in a host window. 'Cut' and 'paste' do not communicate with the host. *Data transfer between windows is monitored to ensure that it conforms to the SV/MLS mandatory access control policy.*

- Window Management — Windows may be reshaped, moved, put on the bottom or top of the window stack, made current, or deleted.

- Character Fonts — Three fonts are built into the 630 MTG Terminal firmware; they are small, medium, and large constant width fonts. Additional fonts can be downloaded. The number of such downloaded fonts is limited only by memory size. *Only trusted fonts are downloaded; the font downloading program is trusted.*

- Mouse-based Menu-driven Interface — Terminal functions are controlled using menus and a three button mouse. The mouse interface is available to downloaded programs as well as to the firmware.

- Function Keys — There are 14 function keys, 8 of them programmable. Programmable function keys are programmed using the PF-edit window or via escape sequences sent to a host window. *Only escape sequences sent to a window whose label equals the users login label are permitted to modify the PF keys. Programmable function key contents are not preserved after termination of the session.*

- Local Processing — Software developed for the 630 MTG terminal may be downloaded into a host window and executed in that window by the terminal processor. An application that does not require communication with the host may be disconnected from the host and continue running in the terminal as a local process. *All downloaded software must be trusted; the downloader must also be trusted. Downloaded software must conform to MAC labeling constraints. It is the responsibility of the system administrator to verify that downloadable software is trustworthy. The 630 MTG package includes a trusted downloader.*

- Escape Sequences — Like most terminals, the 630 MTG terminal responds to a set of escape sequences that are needed for various display and control functions. Most escape sequences have no security relevance; they control cursor movement, insert characters, scroll up or down, etc. A few escape sequences are more powerful and their functions are potentially security relevant. These escape sequences have been disabled or restricted as listed below:

  — The escape sequence that writes a user supplied string in the label bar has been disabled since the text area of the label bar now contains the security label of the window.

  — The escape sequences that control printing on a printer directly attached to the terminal have been disabled.

  — The escape sequence that redefines the <Enter> key has been disabled.

  — The escape sequence that modifies the character strings associated with the PF keys is only enabled when sent in a window that is operating with the user's login label.

  — The escape sequence that programs a menu entry on the Button 2 Windowproc menu is only enabled when sent in a window that is

operating with the user's login label.

3.1 Window Labeling

Each window, host or local, is labeled with a security label. Host windows are initially labeled with the level and categories at which the user logs in. Local windows will be labeled with the label of the host window from which they are peeled.

A host window is relabeled whenever the host process controlling that window changes level or categories. Thus the security label of a host window is always identical to the security label of the pseudo-tty through which it is communicating with the host. Security labels of local windows cannot be changed. Changing the security label in one host window does not affect the labels of any other windows. Label integrity is assured by preventing users from writing user-supplied labels in the label bar. Label change commands are only allowed from trusted code; they are communicated to the terminal via the trusted channel 0. Whenever the new security label does not dominate the old label the data currently in the window, and in its scrolling buffer, is cleared.

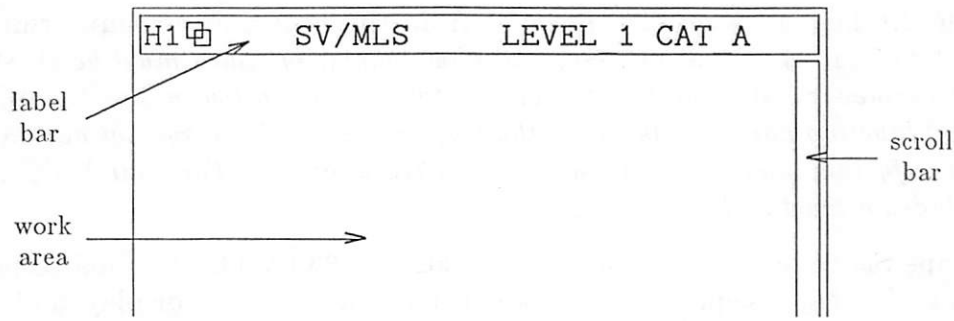


Figure 2. Window Labeling

Each host and local window displays the human readable form of its security label at all times (see Figure 2). When a window is moved, reshaped, or brought to the foreground the label is redisplayed. If the human readable form of the label is too long to fit inside the label bar as much as will fit is written, followed by an indication that the security label as written is incomplete. The minimum size of a window is large enough to always accommodate the level designation, so that only category information can fail to be displayed. The full label is always available via a pop up window.

3.2 Secure Login

In the UNIX operating system the process that detects initiation of a login on a port is called a *getty*. This process issues the `login:` prompt, reads the login id, and exec's another process called *login*. *Login* requests the user's password and validates the user's request for service. If permission is granted to the user to access the system a shell is forked. This is the user's "login shell"; it controls

the user's terminal session, providing services as requested by the user. In System V/MLS, the *getty* also polices the environment to ensure that no *login* Trojan horses are on the tty line.

The window labeling requirements constrain us to maintain a trusted communication channel with the 630 MTG terminal after the login process is complete. It is therefore not possible for login to turn the terminal over to a login shell before *layers* is executed. It is also necessary to ensure that the firmware modifications are downloaded into the terminal before control of the terminal is handed over to the user.

For maximum security terminals are hardwired to the host and the type of terminal attached to each tty line is entered into a device clearances database. For each tty line attached to a 630 MTG terminal the device clearances database also contains an entry specifying that */usr/bin/630init* is to be executed in place of the login shell. */usr/bin/630init* performs some terminal verification, described below, then directly exec's *layers*. *Layers* runs a login shell in each window it creates. It is the responsibility of the system administrator to set up and manage the device clearances database using the commands provided in SV/MLS for that purpose.

3.3 Terminal Integrity

In addition to downloading the firmware modifications and *exec*ing *layers*, *630init* ensures that the 630 MTG terminal is in the expected state before allowing the user access to the system through it. The actions to be performed by *630init* are:

- Query the terminal to determine that it is running the correct firmware release.
- Download a terminal checking program into the terminal.
- Compare the checksums reported by the terminal checking program with the expected values. If the values match, continue; otherwise, report failure in the Security Audit Trail and abort the login sequence.
- Download the firmware modifications.
- Exec *layers*.

The terminal checking program computes a checksum of ROM and of the RAM vector table. This further verifies that the expected firmware version is running in the terminal, and determines that these firmware functions are the ones accessed through the vector table. The terminal checker also verifies that no additional firmware has been loaded into the terminal, and that it is, itself, downloaded at the address it expects. Together these checks provide a high level of confidence that the terminal has not been tampered with. The takeover download itself reinitializes terminal memory above the downloaded program as

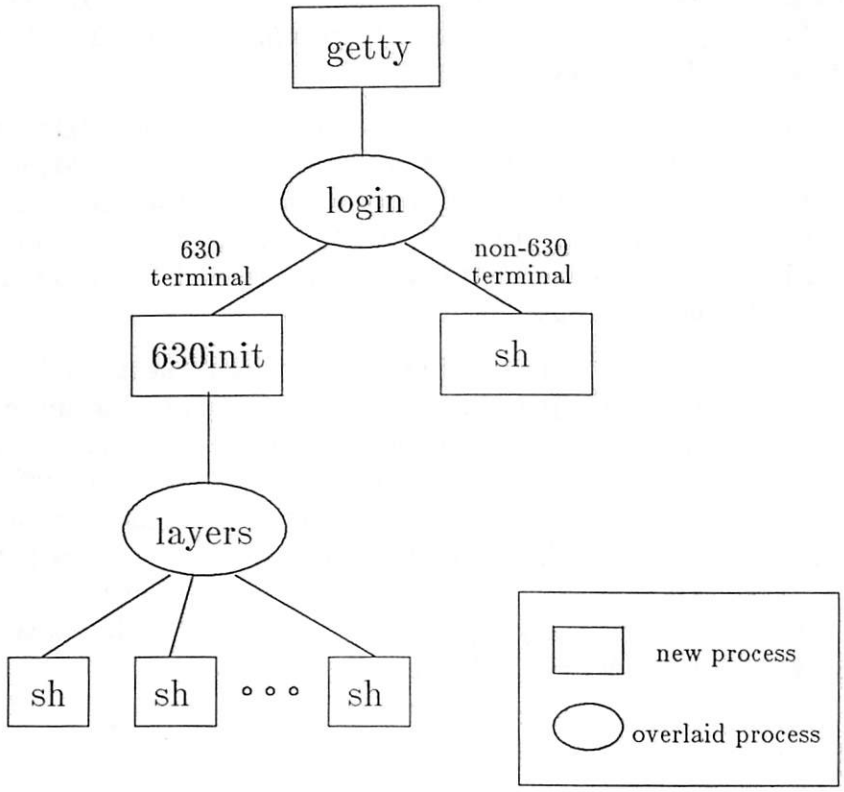


Figure 3. 630 MTG Terminal Login Procedure

a fresh *malloc* arena.

Secure operation requires that the terminal be attached to only one host, and not be connected directly to a printer. The terminal checking program verifies that these limitations are obeyed, in addition to computing and reporting checksums to *630init*. We rely on physical security of the terminal to prevent firmware modifications which would evade the checks supplied by *630init*.

Since the 630 MTG terminal provides scrolling, there is a possibility that a user who neglects to turn off the power to the terminal on logout might leave sensitive information accessible. To prevent such information from remaining in the terminal, all windows, including local windows and the buffers associated with them, must be deleted when the host connection is broken. When the exit menu item is chosen or the terminal detects disconnection of the host all RAM past the SV/MLS modifications will be cleared, the terminal selftest program will be run, and the terminal will be rebooted.

The character strings associated with the user-programmable function keys (PF keys) were previously stored in the 630 MTG terminal in non-volatile RAM. Since the terminal cannot be permitted to retain downloaded information between terminal sessions, at login a new PFkey storage area is allocated in RAM. Any information in the original PFkey area is inaccessible while logged in

to the host; any modifications to the PFkeys do not remain after logout. The original PFkey storage area in NVRAM is still accessible when the user is not logged in to the system. In effect, there are two entirely separate PFkey areas: the volatile one accessible while logged in, the non-volatile one accessible when not logged in.

At login the volatile set of PF keys is labeled with the users login security label Mandatory access controls are applied to data transfers between the PF-edit window and other windows on the terminal. Programming of PF keys via escape sequences is permitted, but only in windows whose security label is dominated by the label of the PF keys. (In effect, PF keys are only able to be modified from windows with the user's login label.) Manual editing of PF keys is allowed, as in the unmodified 630 terminal. A mechanism is provided for downloading PFkey programs during login. The non-volatile set of PFkeys can only be edited manually.

3.4 Control of Downloading

Downloaded programs present a number of special security hazards on the 630. For this reason any downloadable program must be considered to be part of the Trusted Computing Base. We do not permit users to develop and download their own programs for the 630 MTG Terminal. Inclusion of downloadable programs in the "approved" collection is the responsibility of the system administrator. To implement this policy, the *ioctl* that controls downloading has been restricted to execution by trusted processes. The downloader supplied with 630 MTG is trusted and will only download programs that reside in a specified, protected directory.

4. LESSONS LEARNED

Paradoxically, the 630 MTG terminal, as modified for SV/MLS, provides more security than all but the dumbest of ordinary terminals. For instance, any terminal with a scroll memory has more potential to reveal sensitive information after a user has left the workstation than the modified 630 MTG terminal has. Even sending a clearscreen sequence at logout will not prevent an ordinary terminal from retaining information in case the connection to the computer is severed, whereas the 630 terminal can detect the broken connection and clear the screen itself.

Because we do not permit users to change the label at the top of a window, the SV/MLS version of the 630 MTG terminal has trustworthy security labels on windows. This means that a user who attempts to change his or her security label during a login session can verify that the label change took place as requested. A spoofing program cannot catch the command to change the label and fool a user into entering sensitive information into an unclassified file.

Since the 630 MTG modifications permit us to maintain a trusted comunication channel between the terminal and the host for the entire login session, we expect it to be relatively easy to implement a B3 trusted path on the 630 MTG

terminal. A prototype B3 trusted path is already running on our development machine. Such a trusted path permits the user to know that he or she is communicating with the Trusted Computing Base and not a Trojan Horse program when sensitive commands, such as *passwd*, are being executed.

The restrictions on downloading code to the 630 derive from two considerations: the need to extend security labeling to downloaded programs and the need to ensure that the terminal's lack of memory protection is not used to subvert the security policy. It has proven to be relatively convenient to include security labeling in downloaded applications, requiring only about a dozen lines of code. The lack of memory protection on the 630 has proven to be a more difficult problem. In a standard UNIX environment, the actions of a single user on a 630 terminal cannot violate the protections of the host. However, in a multi-level secure environment the lack of memory protection on the 630 can be used to bypass the mandatory access controls enforced by the system. The main reason for restricting downloads is to prevent such violations from occuring. It becomes the responsibility of the system administrator to examine applications proposed for downloading to insure that they access terminal memory only in allowable ways. It would be much preferable to have a hardware memory protection mechanism.

Except for disabling downloads of user-developed applications and not permitting users to modify the labels on a window, we have been able to retain all the capabilities of the 630 Multi-Tasking Graphics terminal, subject only to the restrictions imposed by the SV/MLS security policy and the "Orange Book" B1 security requirements. At the same time we have been able to provide more security than is provided by an ordinary "dumb" terminal.

5. FUTURE PLANS

The version of the trusted 630 MTG terminal described in this paper is the release 1.0 version. Current plans for the 1.1 version, to be available early in 1989, include re-enabling of local printing with trustworthy security labeling of output. We are currently developing a B3 trusted path for communication between the user and the host, and menu selection of operating label in each newly opened window. Such a terminal, attached to a System V/MLS host over a secure connection, such as that provided by an AT&T STU-III secure voice and data telephone, will provide users with a powerful, flexible, secure computing environment accessable via dial-up lines.

References

AT&T87a "User's Guide 630 Multi-Tasking Graphics Terminal", AT&T Teletype Corporation, March 1987.

AT&T87b "630 MTG Software Development Guide", AT&T, 1987.

AT&T87c "630 MTG Software Reference Manual", AT&T, 1987.

AT&T88 "AT&T System V/MLS User's Guide and Reference Manual", AT&T, 1988.

BERN87 Bernet, Y and McCarthy, J.M., "The 630 Multitasking Graphics Terminal", AT&T Technical Journal, Vol 66, Issue 6, November/December 1987.

DOD83 "Department of Defense Trusted Computer System Evaluation Criteria", DOD, August, 1983.

FLIN88 Flink, Charles W. and Weiss, Jonathan D, "System V/MLS Labeling and Mandatory Policy Alternatives", AT&T Technical Journal, Vol 67, No 3, May/June 1988, pp. 53-64.

MCIL88 McIlroy, M.D. and Reeds, J.A., "Multilevel Windows on a Single-level Terminal", Conference on UNIX Security, USENIX Association, Portland, Ore., August 1988.

Secure Window Systems for UNIX

Mark E. Carson, Wen-Der Jiang, Jeremy G. Liang,
Gary L. Luckenbaugh, Debra H. Yakov

IBM Corporation
708 Quince Orchard Rd., Building 884
Gaithersburg, MD 20878

Abstract

Multilevel window systems are a natural way to look at multilevel data. This paper describes some of the functional and assurance aspects in the design of one such system, CMW Windows, which runs on the IBM PC AT and PS/2.* The security functionality of CMW Windows is ambitious, including two parallel labeling systems, one with down-to-character-level granularity. The assurance work is no less ambitious, including some effort at addressing the difficult problem of covert channels inherent in multilevel window systems. We then discuss our preliminary work on what should be done in window systems of more complex functionality, such as the X Window System.*

Introduction

In the past few years, there has been a great deal of interest in secure operating systems, particularly UNIX* and particularly in terms of meeting requirements such as the Department of Defense's *Trusted Computer Systems Evaluation Criteria*[1] (referred to as the "Orange Book") and the Defense Intelligence Agency's (DIA's) *Requirements for System High and Compartmented Mode Workstations*[2] (referred to as the CMW Requirements). Beyond the base operating system and utilities, making a secure system useful requires treatment of other areas as well, such as window systems. In this paper, we describe our experiences in designing and implementing a secure window system for the Secure XENIX* Compartmented Mode Workstation, CMW Windows. We then look at what possibilities other window systems present, and how the work in securing them will change as they grow in capabilities.

Disclaimer

The views expressed in this article are those of the authors and do not necessarily reflect those of IBM.

Functionality and assurance

There are two aspects to consider in designing a secure system, functionality and assurance. Functionality for secure systems consists of such items as tools for doing administrative tasks, and security labeling and its corresponding mandatory access policy. Assurance comes from such areas as system architecture, covert channel handling, and security testing. In a sense,

---

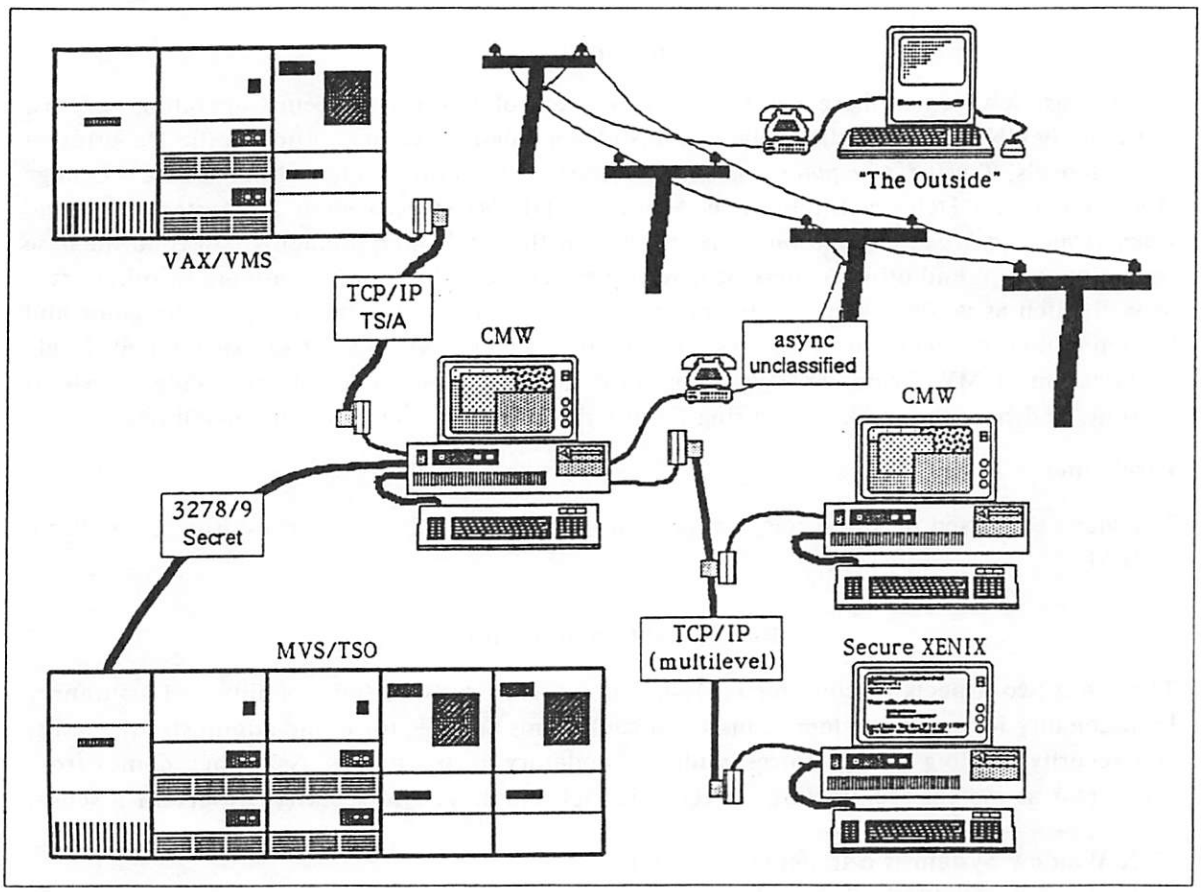
* X Window System is a trademark of MIT.
UNIX is a trademark of AT&T.
XENIX is a trademark of Microsoft.
PC AT, PS/2, MVS and TSO are trademarks of IBM.
VAX and VMS are trademarks of Digital Equipment Corporation.

functionality is the positive side of security, helping people to do the right things. Assurance is more the negative side, making sure people won't do the wrong things, either inadvertently or on purpose.

Although assurance, in the sense of stopping Trojan horses and other such devious attacks, is the most talked-about aspect of security, the functionality provided by the system is no less important. Most computer security problems have occurred not from devious attacks, but from careless or deliberate violations of existing policies.[3] While system designers cannot directly control users' behavior, they can at least endeavor to provide a system which is sufficiently easy and natural to use in the right, secure, way that users will prefer to do it that way.

A good example of that kind of design is a window system which reflects the multiple security levels of work being done on the system. In a typical environment for a secure workstation [Figure 1], information is being received from a number sources with a variety of security levels. In reviewing this information, and perhaps preparing and transmitting reports based on it, the user can do his or her job much more readily if there are no unnecessary barriers to viewing the information freely, and if the system helps the user in labeling extracted reports properly. A multilevel, label-conscious window system provides exactly this sort of functionality.

Figure 1. Secure system environment



In this paper, while attention is paid to the necessary assurance, the main focus is on the functionality needed to allow users to work with multiple levels of data in a reasonable way.

History

While a number of secure systems have included at least minimal windowing functionality, the first effort we are aware of to produce a true secure multilevel window system was done at Mitre as part of a prototype implementation to demonstrate the feasibility of the concurrently developed CMW Requirements.[4] The Mitre prototype, implemented on a Sun 2 system running SunWindows*, introduced two key concepts for multilevel window systems — the window as the unit of sensitivity (mandatory access control or MAC) labeling, with a separate system of labels, called information labels, providing finer granularity of labeling (at the line level in the Mitre prototype) and using a dynamic method of combination ("floating") to label aggregates of data.[5] [6]

We constructed a CMW system as an extension of the Secure XENIX project. The base of the window system is Viewnix*, a kernel-based text-only window system for XENIX, implemented as a new console device driver.[7] [8] Our principal innovation in window system security functionality was a finer-grained label-handling system (potentially at the character level), along with a visible means of indicating it.

Another system, similar in several respects to the CMW, has been built at AT&T Bell Laboratories.[9] [10] It has a single "floating MAC" label system, which combines some features of the CMW's MAC and information labels. It also includes a multilevel window system implemented for the Teletype 5620 intelligent terminal.* Because of the greater restrictions required for MAC labels, the entire contents of a window are at a single MAC level; there is no finer granularity of labeling inside a window.

The CMW and information labels

Since most discussions of computer security center around the Orange Book requirements, some background on the special requirements for the CMW is in order. In Orange Book terms, the CMW requirements are essentially at the B1 level of assurance, but require security functionality beyond anything stated in the Orange Book. The major new features are a multilevel window system as the primary user interface, and a system of information labels in addition to the sensitivity labels of the Orange Book. The window system will be discussed in detail below; here is a brief discussion of information labels.

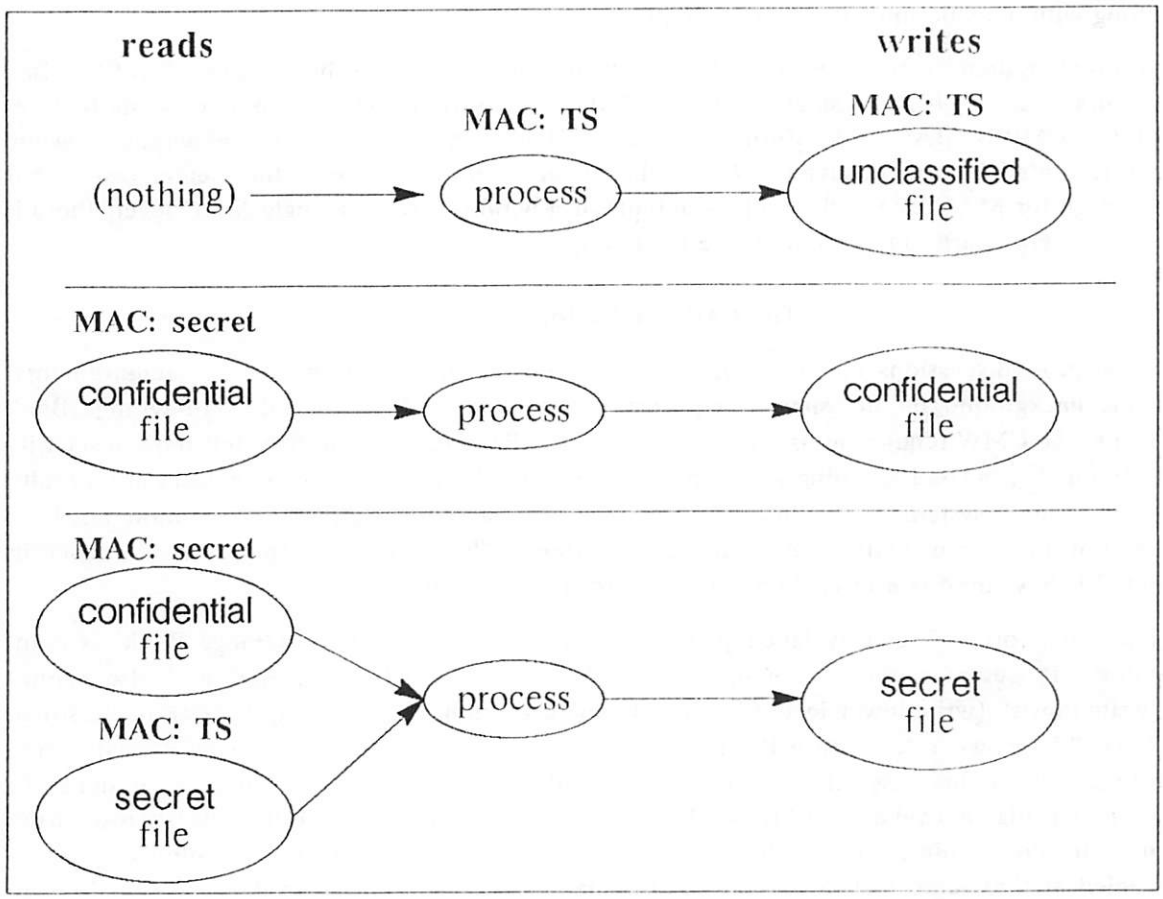
The usual form of security labeling, the "sensitivity labeling" of the Orange Book, is mandatory. Processes cannot "read up" (read files with higher-level labels), and also cannot "write down" (write lower-level files; this latter restriction prevents unauthorized declassification). These two rules ensure the protection of sensitive information, but also create some inflexibility in their use. Because of the no read up rule, users have a natural tendency to work at as high a level as possible, so they can at least see everything. But then, because of the no write down rules, all the files they create, no matter how trivial the contents, must be labeled at that same high level. Since ordinary users are not authorized to change the sensitivity labels on files, they cannot correct the labeling after the fact either, except by appealing to a system security administrator.

---

* Sun 2 and SunWindows are trademarks of Sun Microsystems.
Viewnix is a trademark of Five Paces Software.
Teletype is a trademark of AT&T.

The rules for the behavior of sensitivity labels are necessary for mandatory access controls to work — that's why they are called mandatory, after all. But to help the user avoid overclassification, to help in labeling released documents properly at least, it would be desirable for the system to give some indication of the "true" sensitivity level of data. The CMW provides this through a separate system of information labels. Information labels are both user-controlled and system-controlled; users set them originally and may change them as needed (a variation of the system discretionary access policy says who may change labels and how), while the system updates them through propagation or floating; as a process reads sensitive data its own (process) information label floats to the maximum (least upper bound) of the labels of all the data it has read; when it subsequently writes to files or other objects, their information labels normally float up in a similar fashion, under the assumption they could be receiving the sensitive data [Figure 2]. The sensitivity label then functions as an upper bound on the information label.

Figure 2. The floating process



Note that the goals for the two label systems are quite distinct; information labels are advisory only, but should be as accurate as possible, while MAC labels must be enforced in every way possible, but may be higher than necessary. The desire for accuracy means in some cases objects will have multiple information labels, each applying to some subset of the data in the object.

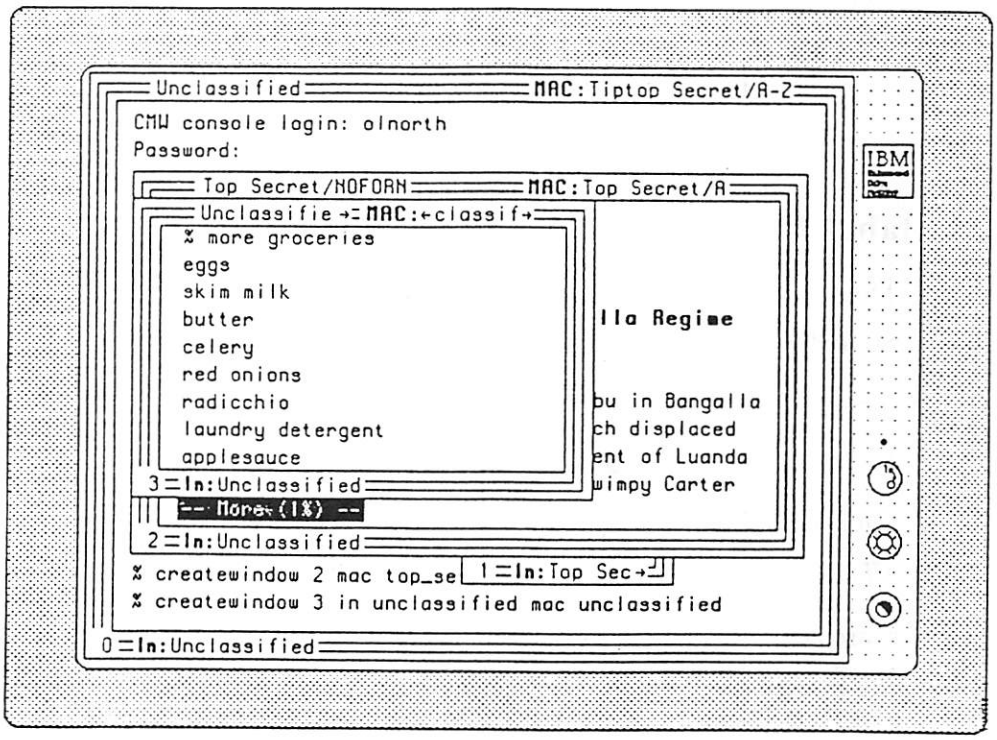
CMW Windows functionality

We will now describe some of the interesting aspects of the work we did for CMW Windows. We will divide the discussion into the user-visible functionality and the underlying assurance.

Sensitivity (MAC) labeling

Mandatory access control handling in CMW Windows is reasonably straightforward. The sensitivity label for a window is specified at the time of its creation, and is thereafter fixed. All processes running in a window are at that window's MAC level. Users may create new windows at any level permitted to them at any time; there need be no relation between the level of the window created and that from which it was created. The window's sensitivity label (along with its information labels) is displayed in the border area around the window [see Figure 3].

Figure 3. Window Appearance



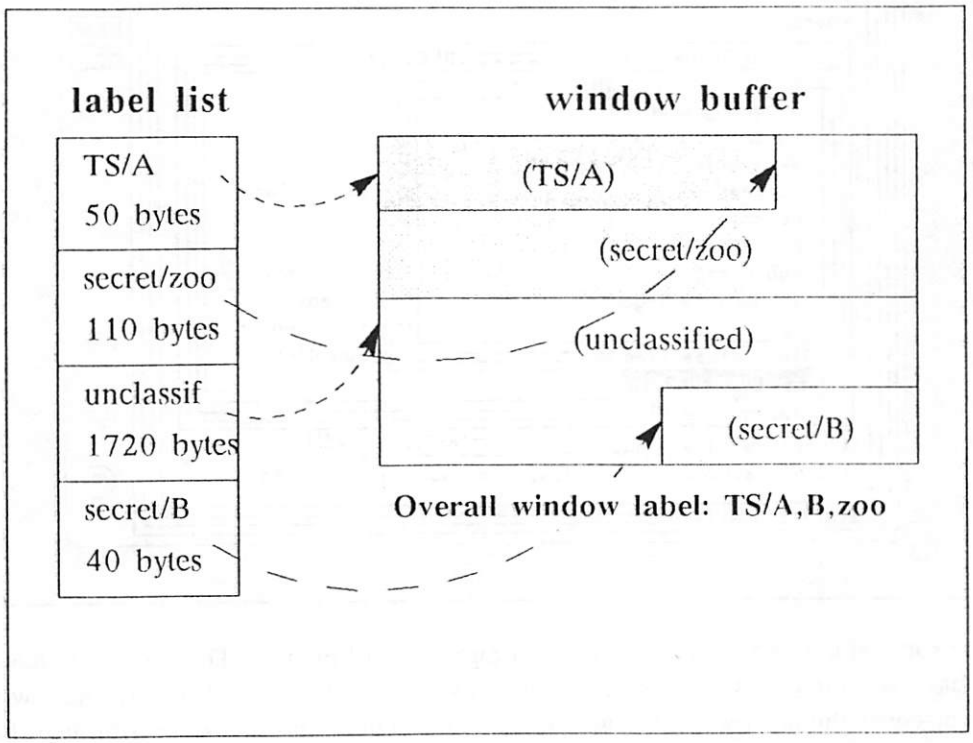
There are some slight complications to this straightforward picture. The sensitivity label for a window has meaning for as long as the window exists, but the (kernel-based) window system can only discover the label when the window is first opened, and the *inode* is brought in core. This means there is a small gap between when the window is "made" (a buffer is allocated for it) and it can be finally labeled. During this period, the window system uses the sensitivity label of the process which made the window as the window sensitivity label. (While the window at this point can contain no data, it still has control information associated with it and hence must be labeled.) The window is normally first opened by the (privileged) command *winlabel* which sets the final window sensitivity label. If some other (nonprivileged) process first opens the window, it must have been at the window's "made" level, which now becomes

its final level. This provides backward compatibility with non-secure Viewnix applications; the windows they create will all be at a single level.

Information labeling

The information labeling system in CMW Windows is more ambitious. Since the charter for information labeling is to assist the user, and since information labels are not subject to the same constraints as sensitivity labels, the window system maintains them with a much finer granularity. For each window buffer (the "backing store" for the displayed window contents), it maintains a corresponding list of block-by-block information labels. The least upper bound of all these labels is displayed in the window border area. [See figures 3 and 4.] The blocks are arbitrary byte ranges which may be as small as a single character. (Of course, it doesn't really make sense to talk about the label on a single character — a (text) word is the smallest conceivable limit — but the window system deals with data at the byte level.) As new data with various information labels is written to the window, perhaps overwriting or scrolling away old data, the label arrays are updated and the overall window information label is recalculated and redisplayed.

Figure 4. Label lists and window buffers



Since in the usual case new data is written to the bottom of the screen while old data scrolls off the top, we optimized for this case by implementing the label lists as an array, maintained essentially as a queue. However, since each window emulates an ANSI-standard terminal, we must also handle writes to arbitrary locations, clearing, insertion and deletion and so on. In other words, we must be prepared to perform full terminal emulation on the label lists as well. Fortunately a number of simple optimizations, such as short-circuiting the overall window label calculation when a maximum is reached, suffice to keep the work required to a reasonable level.

Aside from calculating the overall window information label, the label lists are also used to calculate the labels on data moved by cut-and-paste and block reads (reads from the window buffer). To make the individual labels visible to the user (and originally to debug the label system), a "label palette" feature was added, whereby users can specify what attributes (foreground/background color combinations) should be used to display data with specified labels.

Windows and terminals are unusual in that they have a separate input information label, which is the label applied to keyboard input. In windows, the input label is also displayed in the border area [figure 3]. This label may be changed at any time by the user; CMW Windows provides a simple visual editor for this purpose. (Unfortunately, the design of the window manager means this editor is part of the keyboard interrupt handler!)

An interesting difficulty arises in that, for compatibility with ordinary XENIX, the window system passes all data through the standard line discipline and clist mechanism.[11] The buffering provided by the clists can make it difficult to do fine-grained labeling. For example, if data with one label is being buffered while the user types with a different input label, the user's keystrokes would normally be mixed in with the buffered data, making it difficult to label the two properly. In this case, the clists are simply bypassed; this makes for noticeably less delay in keyboard echo but is otherwise compatible with XENIX handling. However, if output is being stopped (with ^S) while several processes with different information labels write to the window, there is no simple bypass. Here we elected to allow the data label (which is essentially the output queue label) float to the least upper bound of all the data waiting in the clists; when output is resumed (with ^Q), it will be so labeled.

A similar complication on the input side is that Viewnix, like some other UNIX systems, allows processes to write to the standard input of a window — that is, emulate a user typing input to the window. Here, if several processes with several information labels do so simultaneously, or if the user is also typing input simultaneously, data with several labels can pile up in the input queue for that window. If the input is being echoed, each piece of data will be echoed to the window with its individual label; however, when a process finally reads the data, its label will be the least upper bound of all the data in the input queue. Here, the input label is essentially the input queue label.

Trusted path

The CMW has a requirement for a trusted path for performing security-critical operations such as logging in, changing passwords, etc. The user must have some non-interceptible method of requesting this trusted path, and assurance that the path really is trusted; i.e. that programs run under it are also non-interceptible. To implement this in CMW Windows, we used a variation on the method of Secure XENIX.[12] As usual, the trusted path is requested by the so-called Secure Attention Key, which is actually a sequence of two keystrokes, ^Z^Z, typed within one second. This sequence is detected within the line discipline. In Secure XENIX, it then sends a special signal, SIGSAK, to all the processes in the terminal's process group, killing them; this certainly helps make the method non-interceptible but is a little drastic.

For the CMW, we can take advantage of the windowing capability by instead creating a special, isolated window for running sensitive programs. To do this, we have the line discipline instead send SIGSAK to *init*, which creates the desired window. This method was chosen because it is inconvenient to create a window directly from the line discipline, since the key

sequence is detected at interrupt time. *Init* was chosen to create the window simply because it is always available and has sufficient privilege to carry out all the necessary actions.

CMW Windows assurance

The major topic we will address under assurance is isolation: making sure in a multilevel environment that processes in different windows at different levels cannot transfer information in a way contrary to the security policy of the system. For simplicity we will always talk about this as transfering data from higher to lower sensitivity levels, though since sensitivity labels are only partially ordered — the compartment or category field is not hierarchical — we really mean any transfer from level A to level B where A is not $\preceq$ B.

Isolation of window contents

The first and most important aspect of isolation is isolating the window contents themselves. If a lower-level process could directly read the contents of a higher-level window, we would really have no security at all; the breach would have to be regarded as an overt channel, and not a covert one. This seems obvious, but with many systems, user-level processes can directly map video memory. In effect, the window system is then simply a package polite processes should use to deal with the screen properly. Such systems can be dealt with, as we will discuss below, but for CMW Windows this is not a problem, since such direct mapping is not possible. As a substitute, CMW Windows provides a "block read and write" facility for reading and writing directly to the window buffer. Since a window buffer is at a single MAC level, this presents no security problems (and since the interface is through the read() or write() system call, information labeling can be done as well). While necessarily slower than direct memory mapping, we have found in practice it gives very good performance in those applications we have ported to use it. The main problem of course is that porting is required.

Isolation of the trusted path

Another sort of isolation required is isolation of the trusted path. The trusted path is required in two distinct circumstances: before login, so the user logging in can "shake off" any remnants of the previous login session, and after login, so the user can perform critical operations in an assured manner. To help achieve the former, *init*, before starting up a *getty* process for a terminal or the console, calls the vhangup() system call. (*Init* always does this, though the user who wants extra assurance can type the secure attention key to have it done over again.) This call, which comes from Berkeley UNIX, sends a hangup signal to all processes which have the terminal or console open, and cuts off their read/write access to it. For CMW Windows, we extended the call on the console to work on all processes which have any window open. It further removes all windows except the base window 0 (the console window itself), and restores that window to its initial cleared state. This effectively isolates each login session from the next.

After login, as mentioned previously the user can type the secure attention key to create a special "trusted window." The window is trusted because only the trusted process *init* can create it; no ordinary user process can create it in an attempt to fool the user. This restriction is enforced by the privilege mechanism of Secure XENIX[13]: the privilege PRIV_TSH, possessed only by *init*, is required to create the window. *Init* assures the subsequent isolation of the trusted window by making it owned by root with permissions 0600. The trusted shell *tsh* and commands run under it can read and write to the window, since *init* has opened it for them,

but since in the CMW as in Secure XENIX there is no root in multiuser mode,[13] no other processes outside the window can open it.

One final aspect of the trusted path is the window manager. Since it allows such security-relevant operations as setting the input information label, it also must be non-interceptible. For this particular implementation, this is trivial to assure, since the window manager is part of the keyboard interrupt handler itself.

Since the window manager thus trustable, it allows authorized users to downgrade data using cut-and-paste. The data so moved can be relabeled as the user desires.

Covert channels in multilevel windows

The previous sections discussed the assurance needed to handle overt channels, that is normal methods of transferring information. We also made some effort to identify and handle covert channels as well. Covert channels, in Orange Book terms, are nonstandard or indirect methods of transferring information from higher MAC level to lower MAC level processes. Covert channels could be exploited by Trojan horse programs to secretly declassify sensitive data. Because of this possibility, the Orange Book requires, for levels B2 and higher, the identification and, where appropriate, the stopping, delaying, and/or auditing of such channels.

The Orange Book classifies covert channels into two sorts, timing and storage. An example of both can be found in process execution in UNIX. A higher-level process can execute (through exec()) a lower-level executable file, since this action constitutes a (permitted) read-down. However, a lower-level process can now detect indirectly that this has occurred by (forking and) executing the file itself. Since the text is already in memory, the exec() will complete somewhat faster than otherwise. This is a covert timing channel. For the storage channel, recall that writes to executable files which are currently being executed fail with the error return ETXTBSY. Hence, the lower level process can also attempt to write to the file to detect the fact the higher level process is executing it. (The "storage" being exploited here is essentially the incore inode of the file.)

Of the two sorts of channels, timing channels are the more difficult to handle without degrading the performance of the system. In the previous example, the bandwidth of the storage channel can be easily reduced to a negligible value (<1 bps is quite adequate) by delaying (in the kernel) before the error return. (In effect we have converted a storage channel into a (low bandwidth) timing channel.) Since most programs would simply be exiting after such an error, this has no real effect on the normal performance of the system. However, in the timing case, any attempted handling must essentially forego the advantage of shared text. For this reason, and the practical fact that timing channels are generally much more difficult to exploit effectively in real systems, we will ignore them here and instead consider some of the more prominent storage channels.

Multilevel window systems present a rich array of covert channel possibilities. One class is the control information associated with each window, such as its size and position, colors, etc. In CMW Windows, this information can be read and written using special ioctl() calls. Handling here was quite straightforward; each ioctl() was classified as a read- or write-type, and execution of them was restricted to processes which could have read or write access to the window in question. (For compatibility with Viewnix, it was not required the processes actually have that window open for read or write.) In some cases no restriction was necessary. For ex-

ample, the window overlap ordering is write-only (a process can float or sink a window, but can't find out a window's current position in the stack), so no information can be transferred through it. In some other cases, the ioctl()'s affected some shared object, such as the key mapping or the font used. In this case, we simply restricted these ioctl()'s to privileged processes. This doesn't really interfere with the usability of the system, since only a few programs like font editors need access to these objects; these may simply be given the appropriate privilege after review. Similarly, programs which need to use the other ioctl() calls without restriction (an example is the program *winconfig* which gives a summary report of the window configuration) may be given the privilege to do so. In this case, we decided a single privilege, PRIV_WINDOW, sufficed to cover all the privileged window ioctl()'s.

Window systems commonly present a name space covert channel in the resource ids assigned to objects. A simple example in CMW Windows are window ids. This channel may be exploited in two different ways. First of all, processes can request the creation of a window with a specific window id. If that id is in use, an error value (EEXIST) is returned. (This is the same sort of channel as the "text busy" one mentioned above.) One general way to handle this sort of problem is to split the apparent namespace of the objects (so, e.g. each security level can have its own window 1, window 2, etc., with the window system mapping these ids to the real window ids). However, because this method is a bit difficult to implement and introduces some compatibility problems, we did not adopt it here. For this sort of id, it is acceptable simply to delay before return as above.

For other "resources" such a delay may not be acceptable. As an example, consider creating an *xterm* window under the X Window System. *Xterm* must allocate a pseudo-terminal device for communicating with the process running in the window. The accepted way of securing a pseudo-terminal is to try to open the controlling side of each in turn until an available one is found. If each failed open is delayed for a second or two (the typical value to get the bandwidth < 1 bps), it could take up to half a minute to get one. Clearly a more intelligent policy would be needed here.

The reason delay is acceptable for CMW Windows is that the normal way to create a window is not to specify the window id; the system then creates the next available window. Unfortunately, this introduces the other method of exploiting this channel. By examining the id of the created window, the process can find out how many other windows have been created. This channel actually passes several bits at once; for example, if 0–7 windows could have been created, 3 bits are passed each time. (With a bitmapped encoding, 8 bits can be passed each time.) Obviously delaying is not acceptable here. Fortunately, this channel can be (mostly) handled by returning a *random* window id out of the pool of available ones instead of simply the next one.

However, this still does not completely solve the problem. The unfortunate thing about window id's in this context is that there is only a small (15) pool of them available. A higher level process can overcome the randomization by first creating all or most of the available windows and then selectively deleting ones in turn, so that it still effectively determines what window id the lower level process will get. (Here, a clever encoding — selectively deleting 4 windows at a time — can pass around 6 bits each task switch.) But at least at this point, we have a sufficiently unusual circumstance (almost all the windows allocated, and a lower-level process requesting to allocate a window just freed by a higher-level process) that we can introduce delays without fear they would affect the normal performance of the system.

To get an idea of the size of such channels, we measured the bandwidth (amount of information transferrable) of a simple implementation of the last scheme above (a higher-level process selectively freeing a single window which a lower-level process then creates) on a variety of systems [Figure 5]. All systems were otherwise quiescent, and no covert channel delaying was being done. The figures given are for the highest sustainable bandwidths we could achieve. We could briefly get higher rates — up to 300 bits/second on the Model 80 — but this risked the processes frequently getting out of sync.

Figure 5. Bandwidth values for the window id covert channel

System	Bandwidth
PC AT (6 MHz 80286)	50 bits/second
PS/2 Model 60 (10 MHz 80286)	95 bits/second
PS/2 Model 80 (16 MHz 80386 (286 mode))	150 bits/second

One "channel" we have not mentioned so far is far more significant than any of these. Since the window system allows creation of new windows at any level, no matter what the level of the current window, processes could easily downgrade large quantities of data through the arguments and environment passed to commands executed in lower level windows. (Ordinary processes cannot directly create windows at different levels or run processes in such windows, but they can execute the privileged processes which do so.) To prevent this problem, it is sufficient to ensure that lower level windows can be created or used only by direct user intervention.

To accomplish this, a system configuration parameter can be set to make the privileged processes *winlabel* and *runwin*, which label windows and run processes in them, require that labeling or running processes in lower level windows only be done from the trusted shell. Here the isolation of the trusted shell works two ways: on the one hand, the user can be sure in running commands from the trusted shell that the commands he or she selected are in fact performing the tasks he or she requested without making unauthorized diversions, while on the other hand, programs run from the trusted shell can be sure that they were in fact directly requested by the user and are not being used in some covert scheme to break the system's security.

Other window systems

We have done some preliminary work in investigating the implemention of CMW functionality on newer window systems, such as the X Window System[14] and NeWS.*[15] [No product commitment is made or implied.] Here are some observations:

Communicating label information

Since CMW Windows is kernel-based, it is a trivial matter for it to determine the labels for data written to windows and appropriately label processes which read data from windows. However, modern server-based window systems present two additional complications: the window system is implemented as a user-level server process that clients communicate with through IPC's; and the window system is distributed, in that these IPC's may be network IPC's.

---

* NeWS is a trademark of Sun Microsystems.

In this environment, it is necessary to communicate label information in a non-interceptible manner along the data channel. For local communication on the CMW, message queues provide the appropriate functionality; a message queue has a single MAC label, while each individual message has an information label which can be read along with the message. Reading a message causes (non-privileged) processes to float up just as any other read does. Hence, the (privileged) server can determine the appropriate label for data it receives from clients, and can force clients which receive event information to float appropriately. Shared memory, while a more efficient communication means, is less suitable for this purpose since it does not provide individual labeling of data.

For network communication, we have a prototype multilevel TCP/IP implementation[16] which uses the IP security option[17] to transmit both mandatory access labels and information labels at the packet level. For control packets, the MAC label is passed, while for data packets the information label is passed. This provides the same functionality as message queue labeling in the local case. Note that in both cases, message- or packet-level labeling may cause some loss in efficiency if data with varying labels is transferred, since a packet can be no larger than the amount of data with a single information label. However, normally most data will be at a single level.

It is desirable not to require the server to explicitly read the labels of all data it receives. For this reason, a new signal SIGLABELCH can be caught to inform a process that the label has changed on a file descriptor or IPC identifier it is about to read. (The signal is sent at the point of read, not when the information was written.) The process can then read the new label, and proceed with reading the data.

Input labeling

In the discussion of CMW Windows above, and in the CMW Requirements, input labeling was discussed only in terms of keyboard input (or simulated input). But the input label should really be regarded as the event stream label. In this context, we define a new event type LabelChange which may be selected for by applications, to determine changes in the input label. This is somewhat easier to handle than the SIGLABELCH signal, and may be received and interpreted by applications on machines which have not implemented labeling as part of the operating system.

Handling label information

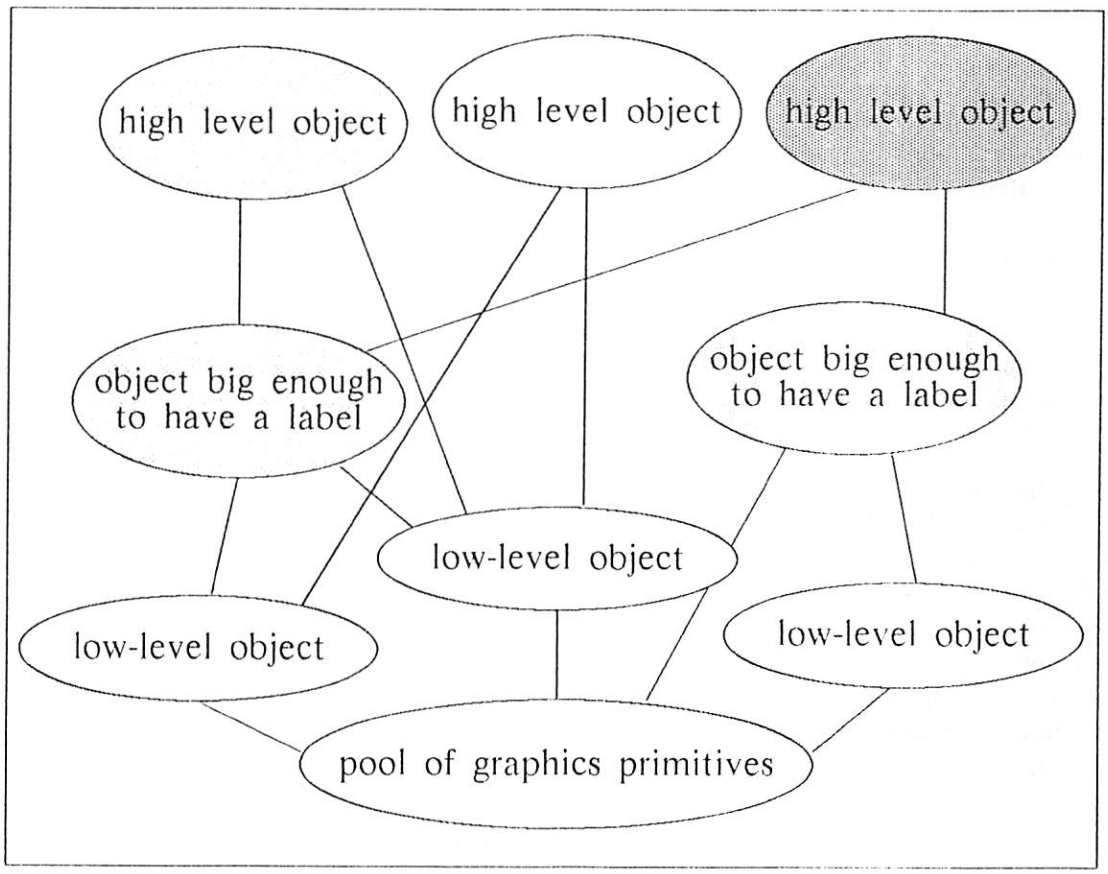
In terms of information labeling, it would clearly be foolish in a graphics window to do the pixel-level analog of the character-level labeling done in CMW Windows. Labeling should really be handled at the object level.

Most graphics primitives (just like single characters) can't be regarded as sensitive. Of the common graphics objects, only text, bitmaps (or pixmaps), and pictures (groupings of lower-level objects) have the possibility of needing labeling. If objects with different labels are combined, the label of the combination is then the least upper bound of the two — that is, the usual floating process is used to determine the new label [figure 6].

This observation leaves open the question of where this labeling should be handled — in the server or the (trusted) application. At the window level of course handling in the server is required, so that it can properly label windows generated by untrusted applications. Whether it should handle labeling with finer granularity depends on whether it is prepared to maintain the necessary information, that is the higher-level structure of the information written to the

window. The recent trend toward extensibility in window systems does provide promise of such a capability. In any case, though, there is still a need for trusted applications to handle such labeling, at the very least for storage purposes.

Figure 6. Objects and labeling



Assurance aspects

We mentioned above that in many systems, user-level programs can actually map video memory and hence read and write it directly. This sort of usage is even more common when dealing with graphics, simply because of the greater bandwidth to the display required. This situation is changing, as server-based window systems, by offering a higher level of interface and the advantages of network connections, can demand the sacrifice of lower bandwidth. But in the short term, can such direct mapping be accomodated in a secure environment?

One approach for doing so is the virtual terminal, such as is implemented in AIX for the RT.*[18] A virtual terminal, or virtual display, appears to the application to be the entire display. Virtual terminals are a sort of low- level or meta-windowing; for example, in AIX each virtual terminal can be running a separate X-Windows session. A virtual terminal manager (which can be separate, as in AIX, or integrated into the window manager) maintains a set of screen images, and decides which of these images are to be currently displayed. In an integrated window/virtual terminal manager with a simple system of job control, we can handle memory mapping with this sort of scheme: if a process requests direct video mapping, a new virtual terminal is created, the current screen is copied to it, the screen (or at least the higher-

* AIX and RT are trademarks of IBM.

level data in the screen) is cleared, and the application is allowed to run. If at any time the user requests to switch to another virtual terminal, or another process, say at a different level, request memory mapping, the application is stopped and the requested virtual terminal is selected. Generally speaking, in switching between virtual terminals only processes with the screen mapped need to be stopped; processes using normal reads and writes to access the window can proceed normally.

Conclusion

A point which has surfaced several times in this paper is that for any restriction imposed by a secure system, there should be some specific way of overcoming it. Thus, when restrictions had to be placed on some window ioctl()'s, a privilege was added to overcome them. When covert channel possibilities disallow the creation of windows at lower levels, the trusted shell can be used to do the task (and in fact the system installer can selectively disable such covert channel restrictions as well). When window isolation forbids direct access to video memory, a virtual terminal approach can be used to allow it. Indeed, the main idea behind this work is that the window system and information labeling system are a solution for dealing with the restrictions of mandatory access control.

This emphasis might seem strange in a discussion of computer security, but in fact it is really something essential. If people are to live with secure systems, it must be possible to make exceptions when needed, exceptions which are no larger than necessary to meet the particular need. Indeed, UNIX presents an excellent example of this, with the setuid/setgid feature provided as a way to get around discretionary access controls. Setuid is often disparaged in discussions of security, but aside from the overloading of privilege on setuid root in UNIX (and the odd bug or two), it really is precisely fitted to its job. In terms of the more severe restrictions presented by mandatory access controls, multilevel window systems seem to be a maintainable level of exception.

Acknowledgements

We wish to acknowledge our indebtedness to John Woodward's group at Mitre, for helping to educate us on the goals for secure window systems and labeling; to Leslie Dotterer and Forrest Stoakes of IBM, for not letting us get away with half-measures; and to Virgil Gligor of the University of Maryland, for sensitizing us to the subject of covert channels.

References

[1] Department of Defense, *Trusted Computer Systems Evaluation Criteria*, DoD 5200.28-STD.

[2] J.P.L. Woodward, *Security Requirements for System High and Compartmented Mode Workstations*, available from Mitre as MTR 9992 (revision 1) and from the Defense Intelligence Agency as DIA report DRS-2600-5502-87 (REV 1).

[3] John Quann, Peter Belford, "The Hack Attack: Increasing Computer System Awareness of Vulnerability Threats," *Proceedings of the AIAA/ASIS/IEEE Third Aerospace Computer Security Conference*, 1987.

[4] P.T. Cummings, D.A. Fullam, M.J. Goldstein, M.J. Gosselin, J. Picciotto, J.P.L. Woodward, J. Wynn, "Compartmented Mode Workstation: Results Through Prototyping," *Proceedings of the 1987 IEEE Symposium on Security and Privacy*.

[5] J.P.L. Woodward, "Exploiting the Dual Nature of Sensitivity Labels," *Proceedings of the 1987 IEEE Symposium on Security and Privacy*.

[6] The label terminology for the CMW changed between the first set of requirements issued in 1986 and the revision issued in late 1987. The cited papers use the old terminology, which should be kept in mind when reading them. Here are the mappings:

Old terminology	**New terminology**
MAC label or level	sensitivity label
sensitivity label	information label

[7] Mark E. Carson, R. Scott Chapman, Wen-Der Jiang, Jeremy G. Liang, Debra H. Yakov, "From B2 to CMW: Building a Compartmented Mode Workstation on a Secure XENIX Base," *Proceedings of the AIAA/ASIS/IEEE Third Aerospace Computer Security Conference*, 1987.

[8] *IBM PC Compartmented Mode Workstation System CMW Windows*, IBM, 1987.

[9] M.D. McIlroy, J.A. Reeds, "Multilevel Security with Fewer Fetters," *Proceedings of the UNIX Security Workshop*, 1988.

[10] M.D McIlroy, J.A. Reeds, "Multilevel Windows on a Single-level Terminal," *Proceedings of the UNIX Security Workshop*, 1988.

[11] Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, 1986.

[12] V.D. Gligor, E.L. Burch, C.S. Chandersekaran, R.S. Chapman, L.J. Dotterer, M.S. Hecht, W.D. Jiang, G.L. Luckenbaugh, N. Vasudevan, "On the Design and the Implementation of Secure Xenix Workstations," *IEEE Transactions on Software Engineering*, Vol. SE-13, 2, February 1987.

[13] M.S. Hecht, M.E. Carson, C.S. Chandersekaran, R.S. Chapman, L.J. Dotterer, V.D. Gligor, W.D. Jiang, A.Johri, G.L. Luckenbaugh, N. Vasudevan, "UNIX Without the Superuser," *Summer 1987 USENIX Technical Conference Proceedings*.

[14] Robert W. Scheifler, J. Gettys, "The X Window System," *ACM Transactions on Graphics*, vol 5(2), April 1986, pp 79–109.

[15] James Gosling, "SUNDEW: A Distributed and Extensible Window System," *Winter 1986 USENIX Technical Conference Proceedings*.

[16] Wilhelm Burger, Narayanan Vasudevan, "Networking of Secure AIX and Xenix Systems," *IBM SID Technical Report TR 85.0076*, 1988.

[17] M. St. Johns, "Draft Revised IP Security Option," *RFC 1038*, January 1988.

[18] *IBM RT/PC AIX Operating System System Reference*, IBM, 1987.

A Trusted Network Architecture for AIX Systems

Chii–Ren Tsai, Virgil D. Gligor†,*
Wilhelm Burger, Mark E. Carson, Pau–Chen Cheng†, Janet A. Cugini,
Matthew S. Hecht, Shau–Ping Lo, Sohail Malik, N. Vasudevan*

IBM Systems Integration Division
708 Quince Orchard Road
Gaithersburg, Maryland 20878

ABSTRACT

We built an experimental prototype of a trusted AIX[1] network based on the DDN (Defense Data Network) protocol suite [DDN 85] and homogeneous experimental AIX Operating System designed to satisfy C2 security requirements [Hecht 88]. AIX is a version of the UNIX[2] operating system on the IBM RT PC[1]. This network contains security features designed to satisfy both the C2 requirements of the *TNI* [TNI 87] and the *TCSEC* [TCSEC 85], and additional security requirements of higher security classes such as a B3 *trusted path* for *Telnet*. We define the Network Trusted Computing Base (NTCB) of the AIX network, present the architecture of network protocols and trusted network applications that comprise the NTCB, provide some design details, and explain how this architecture satisfies the C2 security requirements.

Disclaimer: This paper makes no product commitment, expressed or implied.

1. Introduction

This paper presents the architecture of a secure network using trusted DDN basic protocols (TCP/IP) and network applications [DDN 85] for a version of the AIX operating systems described in [Hecht 88], and that we call AIX' (read, "AIX prime"). The architecture includes the interfaces of the user and administrator and programmer, and the organization of subsystems modules. This architecture is constrained by:

(1) the effective use of the trusted computing base (TCB) of the AIX' Operating System,
(2) the configurable nature of security features in the AIX' Operating System,
(3) the specifications of the basic protocols and network applications,
(4) the available network code, and
(5) compatibility with the existing network applications.

The network security requirements, derived from the the *Trusted Computer Systems Evaluation Criteria* [TCSEC 85] and *Trusted Network Interpretation* [TNI 87], define the basic security requirements for computer systems and networks. Our architecture attempts both to satisfy the C2 requirements and to minimize the number of modifications to the existing network code.

Significant work on network security has been done aiming at satisfying specific TNI or TCSEC requirements, or enforcing network security. This work includes Boeing's MLS LAN [Schnackenberg 87], Secure Xenix[3] Networking [Burger 87] and SDNS [Branstad 87, Linn 87, Nelson 87, Sheehan 87, Tarter 87]. The scope of these projects is different from ours. For example, the first two projects focus on various aspects of non–discretionary access controls of LAN components, whereas the third focuses on the security aspects of communication networks. However, none of these projects address the discretionary access control problems of

---

* This author's address is: VDG, Inc., 4901 Derussey Parkway, Chevy Chase, Maryland 20815.

The first author's network addresses are: uunet!pyrdc!ibmsid!crtsai and crtsai@eneevax.umd.edu.

† This author's address is: Electrical Engineering Dept., University of Maryland, College Park, MD 20742.

[1] AIX and RT PC are trademarks of International Business Marhines Corporation.

[2] UNIX is a trademark of the AT&T Bell Laboratories.

[3] XENIX is a trademark of the Microsoft, Inc. Secure XENIX is an IBM Systems Integration Division product offering (available from July 1987).

existing communication protocols. In this paper, we illustrate some of the deficiencies of standard protocols in discretionary access control and suggest protocol modifications to resolve these deficiencies. We define the Network Trusted Computing Base (NTCB), its interface, and the security policy supported at the NTCB interface. This is more important than it might first appear because security assurance of specific policy support, NTCB noncircumventability, and NTCB tamperproofness cannot be obtained unless the NTCB components and interface are carefully defined. Such security assurance forms the basis for all protocol security properties (not involving communication security properties based on cryptographic mechanisms – discussed in [Voydock 83]). In this paper we also present an implementation of network trusted path in *Telnet* and a mechanism for auditing network events – two of the important accountability mechanisms of trusted systems.

We assume that the readers are familiar with the AIX' system [Hecht 88], and DDN basic protocols (TCP/IP) and network applications [DDN 85].

This paper contains seven sections. Section 2 presents the requirements of class C2 and our architectural assumptions for networking. Section 3 defines the Network Trusted Computing Base (NTCB) and shows the NTCB can satisfy the assumed system architecture requirements. Section 4 describes the NTCB–supported security policy. Section 5 illustrates the application of the security policy described in Section 4 to the AIX' TCP/IP. Section 6 discusses security installation and configuration. Section 7 concludes this paper.

2. C2 Requirements and Architectural Assumptions for Networking

2.1. C2 Requirements for Networking

We extend the TCB (Trusted Computing Base) to a NTCB to satisfy C2 network security requirements. The architecture is designed to satisfy the following TNI and TCSEC requirements:

(1) C2 NTCB Definition
(2) C2 Discretionary Access Control (DAC)
(3) C2 Object Reuse
(4) C2 Identification and Authentication
(5) C2 Audit
(6) C2 System Architecture
(7) "C2" Import/Export Information

and the additional requirement,

(8) B3 Network Trusted Path.

An implicit but necessary requirement is to unambiguously define the NTCB. To define the NTCB, we specify its components, its user and programmer interface, the subjects and objects under its control, and the security policy that defines the discretionary access controls. The structure and components of the NTCB are defined by the functionality it provides, the threats it counters, and the access policy it enforces.

Requirement (2) causes access control to be enforced between named users and named objects as stated in the security policy. Access rights to objects are given on an individual or on a group basis; access permission to an object by users not already possessing access permission is assigned only by authorized users.

Requirement (3) specifies that all previous network access authorizations are revoked from a transient data object before the object can be reused, and that objects are clean at allocation time.

Requirement (4) causes all users of AIX' to be identified and authenticated before they can initiate any actions that are mediated by the TCB. Identification is required to name subjects. Authenticated data is protected from unauthorized modifications.

Requirement (5) causes security relevant events to be audited and associated with the subject initiating it. Security relevant events are user authentication, creation of protected objects, unauthorized access attempts on objects, and so on. Audit data is protected from unauthorized modifications.

Requirement (6) states that the NTCB is noncircumventable, namely that all access to objects is mediated by the NTCB, and that the NTCB is tamperproof.

Requirement (7) states that any exported object (e.g., file) has a correct DAC label (e.g., owner, group, mode), and any imported object is assigned a correct DAC label. By *export* (resp. *import)* we mean objects leaving (resp. entering) the NTCB.

Requirement (8) specifies that the TCB should support a trusted communication path between itself and users for use when a positive TCB–to–user connection is required (e.g., login, change password).

2.2. Architectural Assumptions

We make the following assumptions about network security:

(1) *Only Trusted Systems are Included in the Network.* The network consists of only secure systems. The development is geared towards a network of AIX' systems designed to satisfy C2 requirements. (Untrusted systems can be included in the network, provided that a minimum degree of security is added for import and export of information. This issue is outside the scope of this architecture. Furthermore, the architecture assumes that the physical security of the network components is assured. This means that the security of communication media must be enforced. However, if we allow untrusted systems to be included into the network, then the trustedness of communication media is no longer sufficient to support NTCB protection because the NTCB can be easily circumvented by the untrusted systems. Therefore, the architecture may not satisfy C2 requirements.)

(2) *Only Significant Security Threats are Handled.* Certain security threats such as traffic analysis and denial of service are considered tolerable in environments where C2 systems are used. Nevertheless, denial–of–service and traffic analysis problems caused by breaches of NTCB isolation and noncircumventability are eliminated. (External threats posed by wire tapping or those countered by link encryption devices are outside the scope of this architecture.)

(3) *Only Standard TCP/IP Applications are Adapted for Security.* We provide security facilities for the use of standard application–level protocols.

3. NTCB Definition

This section defines the NTCB components and interface, and explains how NTCB tamperproofness and noncircumventability are achieved.

3.1. NTCB Components

The security relevant portions of a system are collectively referred to as the *trusted computing base* (TCB). Similarly, the *network trusted computing base* (NTCB) is partitioned among the network components in a manner that ensures the access of subjects to objects in accordance with an explicit and well defined network security policy. In addition to the TCB of individual AIX' systems, the NTCB includes part of the services of TCP/IP and network applications. The NTCB consists of *kernel components*, *trusted program components*, and an *NTCB interface* that is used by untrusted user applications. The kernel components consist of the *socket device driver*, the *network device driver*, and the protocol layers up to and including the TCP/UDP layer or the transport layer (in the OSI model). All kernel components consist of trusted code so that mediation of network access cannot be circumvented by using these components. The network device driver can be accessed only by the kernel components implementing the protocol layers or by trusted programs. Figure 1 shows the components and interface of the AIX' NTCB. It also shows the basic protocols that are implemented in the socket and network device drivers. The socket device driver contains TCP and UDP, and the network device driver, *NET0*, contains IP, ARP, GGP, and ICMP. The network packets are processed by kernel components. The *NET0* device driver handles the Internet routing of packets if the system is configured as a gateway. The code that implements these functions is verified and trusted to ensure that packets are not misrouted. Misrouted packets could lead to an unauthorized release of information. All network device driver code is structured so that it follows the guidelines for trusted kernel code (see Sections 3.3.1 and 3.3.2).

Sockets provide a convenient programming interface to the network facilities. In AIX', the socket interface is implemented by a pseudo–device driver. Socket calls are transformed by the socket library into device–driver calls. User applications that require network facilities call upon the routines of the socket interface. All well–known sockets and their associated services,

such as **telnet**, **xftp**, **name server**, and **printer server**, are registered in the file **/etc/services**, which is used to start these services. This file is protected by the NTCB and is not accessible to unprivileged users.

For AIX', the network applications follow the standard protocols based on TCP/IP, and the existing network code is retrofitted to provide the secure network facilities. A major concern of some network applications, such as Telnet and File Transfer Program, is the handling of identification and authentication information. A trusted environment must be provided for these applications to obtain and mediate identification and authentication data, namely a login identifier and password.

Trusted programs perform remote authentication, establish and mediate connections, and provide access to remote services. Moreover, authentication information is supplied to applications in a trusted fashion, for example, through the use of TCB–protected network profiles. This allows the clean separation of the trusted part of an application requiring network services from the untrusted part irrespective of the functionality of the application.

The network daemons, connections, and accesses to security relevant files must be trusted since they are spawned directly or indirectly by **init** and run in user mode to access system services. Several user programs related to network management and network administration are security relevant, such as **hostname, route**, and **netstat,** so that they must belong to the NTCB also. **Hostname** defines the identity of a system, **route** defines packet routing and thus accessibility to systems, and **netstat** gives information about traffic for various communications. Only privileged users can execute **hostname** and **route**.

Applications, which may access a remote system, require identification and authentication. The exceptions are applications that do not access security relevant data, such as **finger**, **host**, **ruptime** and **rwho**. All network applications that satisfy these requirements can be adapted to the AIX' network environment. The application **tftp** has no identification and authentication requirement and, therefore, cannot be included in AIX' when configured for C2 security.

3.2. NTCB Interface

The dashed line in Figure 1 shows the interface to the NTCB. The socket library and trusted applications, which provide the NTCB user interface, are highlighted with boldface

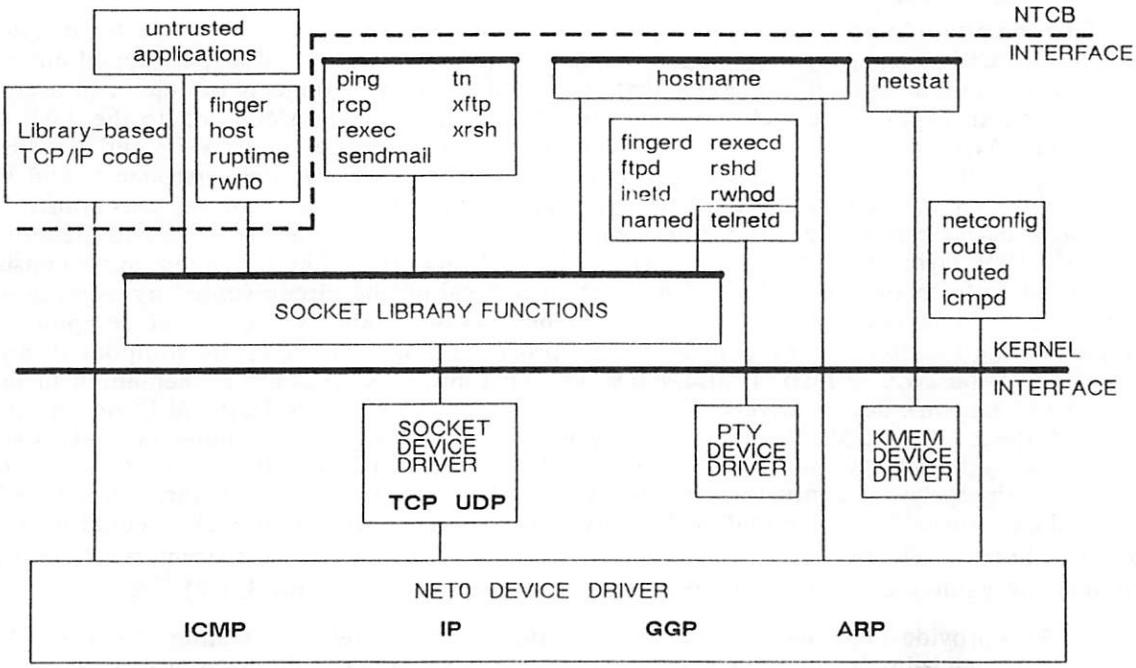


Figure 1. NTCB Components and Interface.

horizontal lines. The NTCB includes the socket-based application layer protocols. For applications that are not network related (e.g., X-Windows), the socket interface is the NTCB interface.

A significant degree of trust in applications that use the socket interface can be obtained by scrutinizing and modifying these applications so that they follow the guidelines for trusted-program development (see sections 3.3.1 and 3.3.2) and by security testing. Socket calls are restricted to programs that execute with the effective group *system* (with group Identifier 0) to prevent unauthorized access to the network. The inclusion of network applications within the NTCB would have the advantage that special provisions for connection establishment would not be required; this approach would be sufficient for the purposes of this work, as the required network applications need not be separated into trusted and untrusted parts. However, this inclusion would add assurance liability because trust would have to be obtained for all applications in their entirety.

There are two approaches for integrating the socket interface in a trusted environment. In the first approach, we begin by assuming that all network applications using the socket interface are trusted. We then relax this constraint and look at the requirements for untrusted code to use the socket interface. We assume that all kernel components are trusted (or that trust can be obtained for them).

The second approach, which we adopt here, allows each user to access the socket interface of the NTCB. This has the advantage that large parts of an application using the secure network facilities can be untrusted, thereby reducing the assurance requirements. The structure of the socket-based network code matches the structure of the NTCB as well. As connections can be initiated by untrusted code, some mechanism is required to confirm that the endpoints of a connection are related and represent the same connection object. In conjunction with the TCP connection mechanism, socket information must be exchanged in a secure fashion. Further, the creation of well-known sockets is limited to trusted programs to avoid the possibility of an untrusted program masquerading as a known service. Network servers, therefore, will always consist of trusted code. This code may invoke untrusted code to do work on its behalf, if it can be shown that the use of untrusted code cannot circumvent the NTCB interface.

The kernel mediates access to the network. There are also constraints on some calls of the socket interface. Calls that change the configuration of protocol layers that are security relevant may be used only by trusted programs.

To take advantage of the NTCB interface, it must be possible to separate an application into a trusted part and an untrusted part. Although this separation can reduce the required assurance work by eliminating the need for producing assurance evidence for untrusted code, it has limited use for network applications that require multiple authenticated communications. The trusted part is required for the authentication of the initial communication. After this is accomplished, control can be given to the untrusted part of the application. Once this is done, however, trusted application code cannot be invoked reliably, as no guarantees can be given (1) that the untrusted code does not invoke a program that intercepts information destined for the trusted program (such as passwords), or (2) that the untrusted program actually calls the trusted program whenever such a call is necessary. Thus, network applications that dynamically require multiple authenticated communications must be trusted in their entirety.

The socket interface is also used by user-level applications that do not necessarily access the network (e.g., the X-Windows interface). Applications that access sockets, therefore, should not be changed by network security requirements. This makes the second approach discussed above, namely that of establishing the socket interface of AIX' as the NTCB interface, the preferred one.

The design of the NTCB does not preclude AIX' from being used in environments where security is not of concern. Untrusted network applications that are built upon the network library, as well as the network library itself, can be offered in such an environment. The code that is not based on socket library functions is called *library based network code*. The configuration of the network code for a secure environment excludes these applications and the network library.

3.3. Tamperproofness and Noncircumventability of the NTCB

Tamperproofness and noncircumventability are the two key properties of NTCB protection and, implicitly, of protocol security. NTCB noncircumventability means that unprivileged users or programs outside the NTCB cannot bypass NTCB interfaces and access protocol and other local or remote objects supported by a network in an unauthorized way. NTCB tamperproofness means that unprivileged users or programs outside the NTCB cannot modify any of the code or data structures of the NTCB in an unauthorized way. The relevance of these concepts can be easily understood by examination of the recent Internet "worm" which afflicted the ARPA network [Page 88]. For example, the **sendmail** attack exploited a debugging feature left in SMTP, which allowed the circumvention of the normal **sendmail** use. Similarly, the **fingerd** attack exploited a flaw which helped modify internal data structures used by kernels of the attacked systems – a violation of tamperproofnes properties. Had careful definition of the NTCB components and interfaces been done, the flaws leading to these attacks would have been discovered. In general, the noncircumventability and tamperproofness properties of the NTCB are important because they represent the *prima facie* evidence that specific security policies can be supported by the NTCB.

Security policies of NTCBs are defined in terms of subjects, objects and access types supported at the NTCB interfaces. The subjects supported by the AIX' NTCB are the AIX' TCB subjects (i.e., processes). However, the *port* is a NTCB object and not a TCB object. Section 4.2 describes a port as an NTCB object and the access control policy applied to ports. The NTCB implements the network security policy. It must be designed in such a manner that the access control and protection mechanisms for the security relevant resources cannot be circumvented. Furthermore, network applications, which are included in the NTCB, follow the guidelines shown below for trusted programs to provide tamperproofness and noncircumventability, and thus satisfy the NTCB system architecture requirement. In addition, the design of the NTCB must also be compatible with the design of the AIX' TCB, so that the security policy in both the TCB and the NTCB is enforced.

3.3.1. Socket Library Functions

The socket library routines of interest are:

accept*, bind*, connect*, gethostid, gethostname, getpeername, getsockname, getsockopt, listen, recv*, recvfrom*, recvmsg*, send*, sendto*, sendmsg*, sethostid*, sethostname*, setsockopt, shutdown*, socket.

The routines marked with an asterisk are security relevant. Modifications to these routines usually consist of restricting their use of certain parameters only by privileged users. If the restriction is not satisfied, then these routines will fail, and **errno** returns with the value **EPERM**.

Trusted code in the kernel such as device drivers must also follow certain programming disciplines to ensure their trustedness. This aids in assurance arguments and in the establishment of test plans. Pointers contained in structures that are passed from the user space to the *NET0* or to the socket device drivers are validated to guarantee that only addresses in the user space are represented.

3.3.2. Trusted Network Applications

Trusted code must not depend on untrusted code to do any work that is security relevant. Trusted code may invoke untrusted code if it can be shown that the use of untrusted code cannot circumvent the NTCB interface.

The shell **/bin/sh** in AIX' is untrusted. This shell, therefore, cannot be used to spawn processes from trusted code. Also, calls to the subroutines **system** and **popen** may not be used as they involve the shell. Programs that use these calls must be modified to fork a trusted program that accomplishes the intended task. Usually, the routines that replace calls to **system** are more efficient as they are tailored to the task at hand. Restructuring of this code does not impact performance adversely.

Guidelines of Trusted Programs

To aid assurance, trusted programs should follow a consistent program structure. This program structure is enforced through code review and configuration management. The generic program structure has four sections:

(1) Section 1 specifies the purpose, interface, environment, special privileges, and auditable events of the trusted program. It also specifies the relevant installation data.
(2) Section 2 defines the data structures used by the trusted program.
(3) Section 3 performs the initialization of the trusted program execution environment in a consistent manner. The initialization consists of closing unneeded file descriptors, and ignoring signals or setting up signal handlers. For a daemon some additional environment initializations are carried out, like disassociating itself from the controlling terminal. Daemon initialization is described in [Lennert 87].
(4) Section 4 performs the specific functions of the trusted program. The program must be a privileged program running with the effective group identifier *system* to make privileged system calls in support of the network security policy. To minimize trusted code the services of trusted programs should be called upon whenever possible.

Programs that need access to resources protected by the real group identifier, but which run with the effective group identifier of *system* to make privileged calls, must be restructured so that network access and access to these resources can proceed appropriately.

3.3.3. Security Relevant Files

Several network security relevant files are updated only by privileged users. These include: **/etc/hosts** defines the mapping between Internet addresses and host names; **/etc/hosts.equiv** specifies remote hosts that are permitted to remotely execute commands on the local host; **/etc/services** and **/etc/protocols** define the services and well-known port numbers as well as the protocol mapping for sockets; **/etc/gateways** defines and maintains routing information; **/etc/net** contains a stanza for each network device (adaptor) that TCP can use which specifies the network address for the system; **/etc/protocols** specifies official protocol names, protocol number and their aliases; **/etc/rc.tcpip** sets up local Internet address and initializes network deamons; **/etc/resolv.conf** defines name server; **/usr/adm/sendmail/sendmail.cf** is the sendmail configuration file, which contains mailer specifications, official hostname, local alias file, and rewriting rules; and **/usr/adm/sendmail/aliases** defines aliases for mailer daemons and for handling mail. These network data files and network configuration files must be owned by group *system* and protected from modification.

The file **/etc/telnet.conf** is used to translate between the AIX' standard terminal names and Telnet terminal names, so that it is not security relevant.

4. NTCB–Supported Security Policy

To satisfy C2 security requirements, we define a security policy and then apply the security policy to the NTCB. The security policy is used to handle security threats. Consequently, protection philosophy is derived from analyzing security threats. Based on the protection philosophy, we specify a practical security policy.

4.1. Protection Philosophy

For the NTCB, we extend the TCB security policy to cover remote users and connections between systems. Access to the network is handled separately from access to a network node.

The following rules state the protection philosophy of AIX' TCP/IP:

(1) **Network Access Rule.** A user authorized to access an AIX' system connected to a network is also authorized to access any system in the network; we call such a user a *network user*.
(2) **Remote Authentication Rule.** A network user accessing a remote AIX' system must be explicitly authorized to do so. The network user is subject to the identification and authentication requirements of the remote system if any data on that system, which is not public with respect to the network, is to be accessed. The communication endpoint on the remote system is identified with the network user and is considered to be a subject for the purpose of enforcing the access policy on the remote system.
(3) **Local Autonomy Rule.** Access to and creation of objects is controlled by the security policy of the site where access or creation occurs.
(4) **Connection Rule.** Access to a connection and its contents is limited to the properly identified and authenticated subjects that created the related endpoints of the connection.

Access to the network is allowed for all users on each site. The Remote Authentication Rule and the Local Autonomy Rule, however, provide the required granularity of a single user accessing another site on the network through a TCP connection. The Local Autonomy Rule states that each site on the network implements its own access controls, and that, at least for the duration of the connection, the access controls on both ends of the connection are satisfied. This rule can be applied for both UDP and TCP connections. The Connection Rule assures that connections exist only between proper parties, that they may be used only by those parties, and that previously existing connection contents cannot be reused.

4.2. Security Policy

The security policy components are (1) discretionary access control, (2) identification and authentication, which includes network–wide trusted path subcomponents, and (3) audit. These policy components refer both to general network access policies and to specific object policies.

4.2.1. Discretionary Access Control

Access to machines and services must be controlled. This requirement implies that a user's ability to access objects and to run programs on a remote system must be controlled. It also implies that user access to protocol services in different layers must be controlled in such a way that lower layers do not subvert the policies of, or the access to, higher layers. Thus, network access control requires both the control of access to user visible objects throughout the network and the tamperproofness and noncircumventability of the protocol trusted–services.

4.2.1.1. NTCB Objects

For AIX', TCP/IP applications are based on the socket–based library functions. Before a connection between two sites is established, a socket has to be created and then bound to a port on each site. Each socket is a user–specific data structure, which defines sending and receiving buffers. From the system point of view, socket structures are the buffers for ports and connections are the means used by ports to import and export information. Connections are *invisible* to users because each connection in the Internet domain is identified by a unique pair of tuples, namely *<IP address, Port>*, for two communicating hosts. A connection for a specific port may be one instance of all the connections associated with that port because different connections can be established in different time intervals. Thus, we cannot view a *socket* nor a *connection* as an object; instead, the *port* can be viewed as an object for TCP/IP in the Internet domain. The object port is used to establish connections. Access to a port from a remote host, namely *remote access*, implies access to the port through its connections and, specifically, the sending and receiving buffers of the sockets associated with the port. Access to a port from the same host, namely *local access*, implies access to its connections with remote ports and their sockets. Access to a port either from a local host or from a remote host through connections must be subject to the discretionary access control policy.

4.2.1.2. Access Control of NTCB Objects

Access to ports is partitioned into *local access* and *remote access*. If we enforce access control on the connections associated with a port, we implicitly enforce remote access control on that port. Control of local access to a port is subject to the control of binding the port to a socket. Therefore, we can state access control of ports by rules for both ports and connections.

4.2.2. Object Reuse

The reuse of a port, which is still bound to a socket, must be prohibited. The reuse of an old connection, which is not completely disconnected, must be prohibited. This is derived from the last statement because, from the port point of view, each connection is only temporarily associated with a port.

4.2.3. Identification and Authentication

Network users and services must be identified uniquely and their authentication must be performed whenever accesses to network services and objects are made. The identification and authentication data must be protected.

4.2.4. Audit

The TNI requires that the NTCB should be able to record the following types of events:

(T1) use of identification and authentication mechanisms,
(T2) introduction of objects into a user's address space,
(T3) deletion of objects,
(T4) actions taken by computer operators and system administrators,
(T5) other security relevant events.

4.2.5. Network Trusted Path

Protocols that implement remote user services without local system mediation must provide the users with a trusted path to remote services.

4.2.6. Import and Export of Information

The policy for import and export of information is enforced in the TCB. We assume that only secure systems are in the network, and thus there is no import and export of information in the NTCB.

5. Application of Security Policy to AIX' TCP/IP

The security policy described in the previous section is applied to the NTCB to satisfy the requirements. This section provides a one-to-one mapping between the security policy and its application.

5.1. Discretionary Access Control

5.1.1. Access Control for Objects

The NTCB object can be accessed either locally by binding a socket to a port or remotely by establishing a connection. A port can only be bound by a socket at a time; this means that local access to a port is limited to one user at a time. After a socket is closed, implicitly the port bound to it is relinquished. Then the port can be bound to another socket. To enforce remote access control to a port, identification and authentication must be performed for each connection. If multiple connections are required for an application, it may not be necessary to perform identification and authentication for each connection; instead, only the first connection need be identified and authenticated. Subsequent connections need only be verified to ensure that they are associated with the authorized user. Thus, discretionary access control for the object in the NTCB can be enforced. Also, the AIX' system has provided discretionary access control for each object in the TCB and the network connection is tamperproof and non-circumventable. Any data access through TCP/IP and network applications is subject to the access control of the site where access occurs as mentioned in the security policy (Section 4.2).

5.1.2. Access Control for Connections

Network applications require identification and authentication for each service that needs a TCP connection. Some applications, such as the File Transfer Program (FTP), may require at least two separate TCP connections at certain times. Identification and authentication of each connection can guarantee the validity of a connection in absence of failures, but would be inconvenient to use. It would be helpful to use some mechanism that would allow multiple connections at the same time and require identification and authentication for only the first of these connections.

Whenever multiple connections are required by network applications, these connections can be associated with the first connection. Consequently, only the first connection requires identification and authentication. Let us consider applications that require two connections, namely a *control connection* and a *data connection*. Control connections transfer and mediate access to remote authentication data and commands, while data connections are used to transfer data. We assume that a *client*, which is the active party, and a *server*, which is the passive party, want to communicate with each other. Thus, the client initiates the control connection, and then the data connection can be initiated by either party depending upon their protocol. Whenever there is no explicit identification and authentication for certain data connections, then these connections must be verified before data transfers start. There are two possibilities for data connections: (1) the server initiates the data connection and (2) the client initiates the data connection. For these two cases, we only discuss appropriate protocols and potential problems for the first case. The second case is similar to the first one.

The Server Initiates the Data Connection.

Figure 2 shows the connection diagram for this case. The IP addresses for the client and server are *IP1* and *IP2*, respectively. *P1*, *P2*, *Q1*, and *Q2* are port numbers. Each connection can be uniquely identified by two tuples, namely *<IP address, Port Number>*; therefore, we use the notation (*<Active IP address, Active Port>*, *<Passive IP address, Passive Port>*) to represent a connection, where the first tuple represents the active site that initiates the connection.

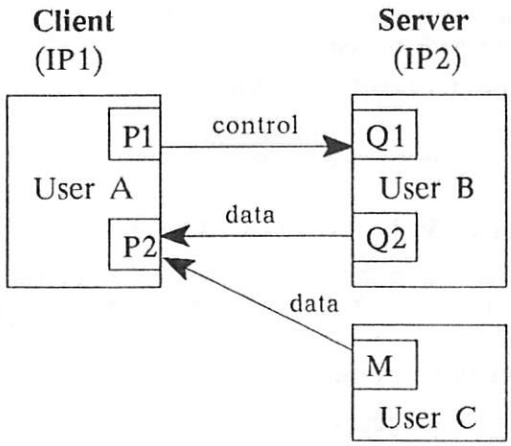


Figure 2. The Server Initiates the Data Connection.

In Figure 2, user *A* at the client site initiates a control connection (*<IP1, P1>*, *<IP2, Q1>*) to transfer remote authentication data and commands. After the control connection is established, authentication must be performed at the server site. Then, user *A* may send a command through the control connection for requesting data transfer and notify user *B* to initiate a data connection to port *P2* at the client site. Thus, user *B* at the server site initiates the data connection (*<IP2, Q2>*, *<IP1, P2>*). If user *A* accepts the data connection without knowing in advance that user *B* is using port *Q2* to initiate the connection, then user *A* cannot verify user *B*'s port number. Since neither verification for its peer's port number nor identification and authentication is performed, a security problem may arise. Suppose a malicious user *C* at the server site knows user *A* is waiting at port *P2* for a data connection from user *B* and initiates the data connection (*<IP2, M>*, *<IP1, P2>*) just before user *B* initiates the connection (*<IP2, Q2>*, *<IP1, P2>*). Thus, only the connection (*<IP2, M>*, *<IP1, P2>*) is established and user *A* starts transferring data with user *C*. This connection violates the discretionary access control policy. Note that the malicious user *C* can be at any site other than the client site.

Solution

To avoid this problem, user *B* must send the port number *Q2* to user *A* before the data connection (*<IP2, Q2>*, *<IP1, P2>*) is initiated. Also, user *A* must verify its peer's IP address and port number after a data connection is established. If the peer's IP address and port number are not respectively *IP2* and *Q2*, then user *A* has to close the connection immediately. This solution must be implemented by *all* user-level applications that *want* to communicate across the network in a secure manner.

Alternatives

In addition to the above solution, there are two other alternatives. Instead of sending the port number for identifying a connection, the first alternative protocol sends the endpoint user identity through the control connection to identify the endpoint user. A user identity is used to identify a data connection; therefore, the user identity must be associated with the connection. Otherwise, the user identity may be misused for an unauthorized connection. Here, a connection would be identified by the endpoint user identities of the connection. For example in Figure 2, the control connection can be represented by *<A, B>*, where *A* initiates the connection. After the data connection (*<IP2, Q2>*, *<IP1, P2>*) is established, user *B* sends its identity to user A to identify the connection. Since user *A* does not know the port for user *B* to initiate the data connection, user *A* merely assumes that the previous connection is initiated by user *B*.

If a malicious user *C* initiates the unauthorized data connection (*<IP2, M>, <IP1, P2>*) just before the connection (*<IP2, Q2>, <IP1, P2>*) could be established, then user *B*'s identity may be used as the identity of the unauthorized connection. Therefore, additional information must be transmitted through the control connection to associate connection and user identity. This solution is more complex than the previous one because the previous solution requires only one exchange of port numbers.

The second alternative is to send the endpoint user identity through a trusted server. This alternative is similar to the first alternative except that it requires an additional trusted server and a different protocol. With the first alternative, standard applications perform identification and authentication through control connections, while this alternative suggests another trusted server to mediate remote authentication data. Thus, the second alternative is more complex than the first alternative. For AIX', connections are trusted because the NTCB is tamperproof and noncircumventable. It is unnecessary to introduce an additional trusted server and/or to use different application protocols.

The above discussion shows that a data connection without separate identification and authentication can be valid whenever an appropriate protocol, such as one of the above suggestions, is used. In AIX', we allow users to create their own services by using socket library functions. The discretionary access control policy can be enforced by private services, which are created by users, if the implementation of these services follows the above protocols to verify the peer's IP address and port number. Thus, users' applications based on socket library functions can enforce the discretionary access control requirement to ensure the security of their communications. Trusted applications, such as FTP, must be verified to discover whether their specific implementations may cause the security problem discussed in the above case.

5.2. Object Reuse

As presented in Section 4.2.1.1, the new object in AIX' TCP/IP is the port. There are *well-known ports* and *regular ports*. The well-known ports are protected by the system and are not directly accessible by users. Regular ports are available to unprivileged users. Both well-known and regular ports have a mutually exclusive property. This means that only one socket can be bound to a specific port at any given time. It is impossible for two sockets, which belong to two different users, to be bound to the same port during the same period of time. Furthermore, a connection is uniquely identified by each pair of *<IP address, Port>*. Since a port cannot be shared by two sockets in the same system for a connection, the connection cannot be accessed in an unauthorized way. If a port is bound to another socket when the current connection is closed, the newly bound socket cannot access a previous connection because the TCP specifies the maximum segment life-time for any connection to be really closed. Thus, a half-opened connection cannot be reaccessed by another socket bound to the same port at the closing site. When a port is reused, it is reset to an initial state so that old connections based on that port can no longer be accessible. Since socket structures serve as the sending and receiving buffers for ports and connections, reuse of socket structures are taken care of by memory allocation and resetting functions, such as **calloc** and **bzero**. These functions can ensure that a newly allocated socket structure does not contain previous information in the same segment of memory. Therefore, the reuse of socket structures, ports, and connections cannot obtain residual information of previous connections.

5.3. Identification and Authentication

5.3.1. Authentication

The identification and authentication of users is strictly based on the mechanisms provided at each site on the network consistent with current usage of the TCP/IP. Every network service such as **telnet** or **xftp**, which requires a TCP connection, performs the identification and authentication before the service starts. Furthermore, because those network services are trusted, the identification and authentication data are well protected and cannot be leaked to unauthorized users. The identification and authentication mechanisms of AIX' also satisfy the identification and authentication requirement for secure communications. To gain access to the network or to access a system on the network, a user must be identified and authenticated by the AIX' system. These mechanisms also uniquely identify a user on whose behalf network services will be performed. Some network services may be offered that do not access security relevant data, such as time servers that supply the time of day. These services are public and may be used without the explicit identification of the requestor on that system.

5.3.2. The .netrc File

Xftp, telnet, and **rexec** provide a convenient option that allows users to write remote authentication data in the file **.netrc** at their home directories. The structure of the file **.netrc** is:

machine *remote_host* login *remote_login_id* password *password*

Remote_host specifies the name of the host on the remote site. *Remote_login_id* and *password* are the user's identifier and password on the *remote_host*. The authentication data, namely users' identifiers and passwords, which are registered and encrypted in file **/etc/passwd**, are the *system authentication data*. Also, users must memorize their own identifiers and passwords, which are the *user authentication data*. In fact, user's authentication data and system authentication data for the user must be the same. From the system point of view, the system must protect only the system authentication data in the **/etc/passwd** file. Users have to protect their own authentication data, since users may store their authentication data in various places, including in system files. Even if users store their authentication data in files, access to these files is subject to discretionary access control and, therefore, users can protect their authentication data. Consequently, allowing users to store their remote authentication data in **.netrc** at their home directory does not violate the security policy.

5.4. Audit

Security relevant network events must be audited and traceable to an individual user. Further, network events that should be audited must be selectable.

5.4.1. Audit Mechanism for the Socket Device Driver

There are six routines in the socket device driver. These routines are: (1)**sockopen**, (2)**sockread**, (3)**sockwrite**, (4)**sockclose** (audited), (5)**sockselect**, and (6)**sockioctl**. In addition, there are twenty-three socket-based library functions in AIX' TCP/IP. Most of these functions invoke **sockioctl** to interface with the socket device driver. These routines and library functions are taken into account for the auditing of the socket device driver.

Criteria for Selection of Audit Events in the Socket Library Functions

Section 3.2.4 shows five types of network events that should be audited. The scenarios for determining specific audit events among socket library functions are presented as follows:

Type T1: Events using identification and authentication mechanisms should be audited in the code at higher layers than the TCP/IP layer. Since the socket device driver supports the layer of TCP/IP, there is no auditing in the socket device driver for this type of event.

Type T2: An unnamed socket structure is created whenever **socket** is invoked. Because each unnamed socket structure is specific to an individual user and invisible to other users, it is unnecessary to audit a **socket** event unless the socket is bound to a name (port). Thus, **bind** is audited instead of **socket**.

Type T3: The deletion of a bound socket is already taken care of by the **shutdown** library function or by **sockclose** in the socket device driver. Both **shutdown** and **sockclose** are audited. There is no other code that deletes a bound socket; thus, there are no other socket deletion events.

Type T4: Usually, actions taken by computer operators and system administrators are audited at higher layers than the TCP/IP layer. For simplicity, however, **sethostid** and **sethostname** are included in the auditing for the socket device driver. There is no auditing for this type of event at the higher layers.

Type T5: Any event related to data access is security relevant. To establish or close a connection and to send or receive data between different hosts are, therefore, security relevant. In the socket library, **accept, connect, send, sendto, sendmsg, recv, recvmsg, shutdown,** and the **sockclose** routine are events of this type. However, **send** and **recv** can only be used when **connect** and **accept** are successfully invoked, respectively. If both **connect** and **accept** have been audited, the auditing for **send** and **recv** is redundant. Therefore, no auditing for **send** and **recv** is necessary.

5.4.2. Audit Events for Application Protocols

The following general events for application protocols and services should be audited:

(a) establishment of a TCP connection
(b) termination of a TCP connection
(c) service request identifying the requester and the service (UDP)
(d) recording of changes to the routing table (**routed**)

5.4.2.1. Telnet

The AIX' provides the Telnet client program **tn** and the Telnet server program **telnetd**. Audit events for **tn** are:

(a) establishment of a Telnet connection
(b) termination of a Telnet connection
(c) failure of a Telnet connection

Each audit record should contain the local port number, remote IP address, remote port number, and the user who is invoking Telnet.

Audit events for **telnetd** are:

(a) invocation of Telnet daemon
(b) termination of Telnet daemon
(c) failure of Telnet daemon

The auditing of **telnetd** has to comply with the auditing of **login** in the kernel to fulfill the completeness of audit. Since Telnet is a function of remote login, the server side must provide the records for different phases of the Telnet. Thus, the audit records for **telnetd** depend upon the audit records of **login** in the kernel for completeness. Each audit record for **telnetd** should contain the remote IP address and the remote port number.

5.4.2.2. Sendmail

Sendmail is a utility for users to send mail to other users both in the same system or in the Internet domain. The file **/usr/adm/sendmail/aliases** may contain aliases for system groups. This file is a security relevant file and has to be protected by the system group (see also Section 3.3.3). The operation to initialize this file has to be audited. In addition, whenever **sendmail** is invoked by any user, the sender and the receiver should be audited. The audit record of sendmail must contain at least the sender, the receiver, and the host of the receiver (if different from the sender's).

5.4.2.3. File Transfer Program

The File Transfer Program consists of the client program **xftp** and the server program **xftpd**. Audit events for **xftp** in addition to the audit events recorded by the socket interface are:

(a) establishment of a control connection identifying the communicating parties
(b) recording the names of files transferred
(c) access failures to remote files
(d) communication failures.

Audit events for **xftpd** are:

(a) the creation of a server identifying the communicating parties
(b) recording the names of files transferred
(c) recording the names of files or directories renamed or deleted
(d) access failures to remote files
(e) the creation of a data connection between two file transfer program servers (third party data transfer)
(f) communication failures.

5.4.2.4. Other Applications

The domain server maps symbolic node names into Internet addresses by contacting systems that maintain mapping tables. The domain server is modified for auditing.

The network facilities of AIX provide three administrative programs: **netconfig**, **netstat**, and **route**. For AIX', **netconfig** and **route** are modified for the generation of audit information. Only privileged users may execute **netconfig** and **route**.

Netconfig processes the file **/etc/net** to add or delete network interfaces for that system. All changes that cause the system to join or leave a network are audited. **Route** manipulates the routing tables. The addition and deletion of routes is audited.

To satisfy the audit requirement, all trusted programs and kernel components of the network facilities are instrumented with auditing capabilities. This requires the definition of all network security relevant audit events and the use of the audit subsystem facilities of AIX' by the network code.

5.5. Network Trusted Path in Telnet

We augment the Telnet programs **tn** and **telnetd** to support a network trusted path. The local secure attention key [Hecht 88] has no effect on any remote system when communicating with the remote system through Telnet. This means the trusted path on the remote system must be requested by a different method from that used on the local system. To support the network trusted path we use the Telnet option mechanism to negotiate the capability of the Telnet server to obtain a network trusted path and define a new Telnet command to request establishment of this trusted path.

Since there is no real secure attention key in Telnet, a new Telnet command is supported which in effect is a network secure attention key. When the new Telnet command for the network secure attention key is recognized, the generic code for the network secure attention key is sent to the remote site. As the Telnet server at the remote site receives the generic code, the server should do whatever is required to establish a remote trusted path; in particular, a remote AIX server will generate the local two-character sequence of the secure attention key and send it to the pseudo terminal device driver to request a trusted path.

The Telnet program is trusted to process the network attention key. The Telnet protocol is extended with a new Telnet option for the network attention key. We define a Telnet option TELOPT_SAK with code 200 and a Telnet command SAK, also with code 200. Using the standard notation of the Telnet protocol [DDN 85], the option negotiation is as follows:

IAC DO TELOPT_SAK the sender requests support

for this option

IAC WILL TELOPT_SAK the option is supported

Whenever a user types "send sak" in Telnet session, the following Telnet command is sent:

IAC SAK

On receipt of this command, the Telnet server **telnetd** passes that system's secure attention key information to the pseudo terminal device driver. This causes all processes that are associated with the pseudo terminal device driver of the Telnet connection to be detached. **Init** then forks a trusted shell for the remote user for the appropriate pseudo terminal. When the trusted shell terminates, **init** again forks a shell (as defined for login) for the user. This mechanism is an extension of the secure attention key mechanism of AIX' to pseudo terminals.

6. Installation and Configuration

6.1. Secure Installation of the NTCB

The names and properties of all programs, libraries, and network data files that are security relevant are added to the file **/etc/security/s_installx**. All network auditable events and actions that comprise these events are added to the file **/etc/security/audit/a_event**.

6.2. Configuring AIX' TCP/IP for Security

By default, the AIX' network is not configured for security; if desired, network security must be explicitly configured at installation time by the system administrator. When configured for security, the AIX' network does not contain **tftp** (no authentication) nor **rlogin** (function subsumed by **telnet**, which has a SAK). Also, the network library-based code cannot be used.

7. Conclusion

The development of secure TCP/IP networks requires significant changes even for the C2 class. We illustrate a variety of protocol-architecture changes necessary for secure C2 networks

based on AIX' systems. The key changes include the definition of the NTCB and its interfaces, the support of a specific discretionary security policy and security relevant audit, and the modifications to the existing protocol code to comply with the guidelines for trusted code in AIX'. The introduction of a trusted path is more important a feature in networking than in stand–alone systems owing to the larger variety of attacks possible in a network. The implementation of discretionary security policies in TCP/IP based networks and the analysis of the NTCB tamperproofness and noncircumventability appear to be significantly more complex activities than in stand–alone systems even when these protocols use a secure operating system base.

Acknowledgments

We thank the following people for helpful discussions and insightful suggestions on the ideas in this paper: C. Sekar Chandersekaran at IBM SID and Ellen Stokes, Nick Camillone, Doug Steves, and Ken Witte at IBM AES. For support of this work we also thank Bob Crago at IBM SID and Gary Snyder, Dan Cerutti, Bill Sandve, Khoa Nguyen, and Bob Willcox at IBM AES.

References

[Branstad 87] Branstad, D., et al., "SP–4: A Transport Encapsulation Security Protocol," Proceedings of the 10th National Computer Security Conference, Baltimore, Maryland, September 1987, pp. 158–161.

[Burger 87] Burger, W., "Networking of Secure Xenix Systems," Proceedings of the 10th National Computer Security Conference, Baltimore, Maryland, September 1987, pp. 254–256.

[DDN 85] Defense Communication Agency, *DDN Protocol Handbook*, December 1985.

[Hecht 88] Hecht, M. S., et al., "Experience Adding C2 Security Features to Unix," Proceedings of the 1988 Summer USENIX Conference, San Francisco, California, June 1988, pp. 133–146.

[Lennert 87] Lennert, D., "How to Write a UNIX Daemon," *;login:*, 12:4, July/August 1987, pp. 17–23.

[Linn 87] Linn, J., "SDNS Products in the Type II Environment," Proceedings of the 10th National Computer Security Conference, Baltimore, Maryland, September 1987, pp. 162–164.

[Nelson 87] Nelson, R., "SDNS Services and Architecture," Proceedings of the 10th National Computer Security Conference, Baltimore, Maryland, September 1987, pp. 153–157.

[Page 88] Page, R., "A Report on the Internet Worm," unpublished report, Computer Science Department, University of Lowell, Lowell Mass., November 7, 1988 (available from the author).

[Schnackenberg 87] Schnackenberg, D., "Applying the Orange Book to an MLS LAN," Proceedings of the 10th National Computer Security Conference, Baltimore, Maryland, September 1987, pp. 51–55.

[Sheehan 87] Sheehan, E. R., "Access Control Within SDNS," Proceedings of the 10th National Computer Security Conference, Baltimore, Maryland, September 1987, pp. 165–171.

[Tater 87] Tater, G. L. and G. K. Edmund, "The Secure Data Network System: An Overview," Proceedings of the 10th National Computer Security Conference, Baltimore, Maryland, September 1987, pp. 150–152.

[TCSEC 85] National Computer Security Center, *Trusted Computer Systems Evaluation Criteria*, December 1985.

[TNI 87] National Computer Security Center, *Trusted Network Interpretation*, NCSC–TG–005, Version–1, July 1987.

[Voydock 83] Voydock, V. L. and S. T. Kent, "Security Mechanisms in High–Level Network Protocols," *Computing Surveys*, 15:2, June 1983, pp. 135–171.